

1. Preface

1. [Itse1359-1000-Preface](#)

2. Language Fundamentals

1. The Basics

1. [Itse1359-1010-Getting Started](#)
2. [Itse1359-1010r-Review](#)
3. [Itse1359-1015-Program Organization](#)
4. [Itse1359-1020-Numbers](#)
5. [Itse1359-1020r-Review](#)
6. [Itse1359-1030-Variables and Identifiers](#)
7. [Itse1359-1030r-Review](#)
8. [Itse1359-1040-Strings Part 1](#)
9. [Itse1359-1040r-Review](#)
10. [Itse1359-1050-Introduction to Scripts](#)
11. [Itse1359-1050r-Review](#)
12. [Itse1359-1060-Syntax](#)
13. [Itse1359-1060r-Review](#)
14. [Itse1359-1065-Visualizing Python](#)
15. [Itse1359-1070p-Preview-Strings Part 2](#)
16. [Itse1359-1070-Strings Part 2](#)
17. [Itse1359-1070r-Review](#)
18. [Itse1359-1080pa-Preview-01-Lists Part 1](#)
19. [Itse1359-1080pb-Preview-02-Lists Part 1](#)
20. [Itse1359-1080pc-Preview-03-Lists Part 1](#)
21. [Itse1359-1080-Lists Part 1](#)
22. [Itse1359-1080r-Review](#)
23. [Itse1359-1090pa-Preview-01-Lists Part 2](#)
24. [Itse1359-1090pb-Preview-02-Lists Part 2](#)
25. [Itse1359-1090pc-Preview-03-Lists Part 2](#)
26. [Itse1359-1090pd-Preview-04-Lists Part 2](#)
27. [Itse1359-1090pe-Preview-05-Lists Part 2](#)

28. [Itse1359-1090-Lists Part 2](#)
29. [Itse1359-1090r-Review](#)
30. [Itse1359-1100-Indexing and Slicing Tuples](#)
31. [Itse1359-1100r-Review](#)
32. [Itse1359-1110-Nested Tuples](#)
33. [Itse1359-1110r-Review](#)
34. [Itse1359-1120-Empty and Single-Item Tuples](#)
35. [Itse1359-1120r-Review](#)
36. [Itse1359-1130-Unpacking Tuples](#)
37. [Itse1359-1130r-Review](#)
2. Control Flow
 1. [Itse1359-1210-The while Loop](#)
 2. [Itse1359-1210r-Review](#)
 3. [Itse1359-1220-Operators](#)
 4. [Itse1359-1220r-Review](#)
 5. [Itse1359-1230-The if Statement](#)
 6. [Itse1359-1230r-Review](#)
 7. [Itse1359-1240-The for Loop](#)
 8. [Itse1359-1240r-Review](#)
 9. [Itse1359-1250-Nested Loops](#)
 10. [Itse1359-1250r-Review](#)
 11. [Itse1359-1260-Loop Modifiers](#)
 12. [Itse1359-1260r-Review](#)
 13. [Itse1359-1270-Functions](#)
 14. [Itse1359-1270r-Review](#)
 15. [Itse1359-1280-Function Arguments](#)
 16. [Itse1359-1280r-Review](#)
3. Classes and Objects
 1. [Itse1359-1410-Overview of Python classes](#)
 2. [Itse1359-1410r-Review](#)
 3. [Itse1359-1420-Understanding self](#)
 4. [Itse1359-1420r-Review](#)

5. [Testing copy of Itse1359-1430-Instance Variables](#)
 6. [Itse1359-1440-Class Variables](#)
 7. [Itse1359-1450-Inheritance](#)
 8. [Itse1359-1450r-Review](#)
 9. [Itse1359-1460-Turtle Graphics](#)
4. File Input-Output
 1. [Itse1359-1510-Reading and Writing Text Files](#)
 2. [Itse1359-1510r-Review](#)
5. GUI Programming
 1. [Itse1359-1610-GUI Programming](#)
3. Change in Presentation Format
 1. [Itse1359-1710-Change in Presentation Format](#)
4. Testing
 1. [Itse1359-1720-Doctest](#)
 2. [Itse1359-1720r-Review](#)
5. Text Processing
 1. [Itse1359-1810-Preface to Text Processing](#)
 2. [Itse1359-1830-String Operations](#)
 3. [Itse1359-1850-Regular Expressions](#)
6. Networking and Databases
 1. [Itse1359-1900-Preface to Networking and Databases](#)
 2. [Itse1359-1910-Networking with HTTP](#)
 3. [Itse1359-1930-Dbm and Shelve Databases](#)
 4. [Itse1359-1950-SQLite Database](#)
7. Putting Python to Work
 1. [Itse1359-2110-Preface to Putting Python to Work](#)
 2. Pygame
 1. [Itse1359-2210-Getting Started with Pygame](#)
 2. [Itse1359-2215-Structure of a Pygame Program](#)
 3. Color
 1. [Itse1359-2220-Color-Introduction to Color](#)

2. [Itse1359-2225-Color-Introduction to Alpha Transparency](#)
3. [Itse1359-2230-Color-Animated Demonstration of Surface Alphas](#)
4. [Itse1359-2235-Color-Animated Demonstration of Per Pixel Alpha Blending](#)

4. Pixels

1. [Itse1359-2240-Color-Animated Fly Through an RGB Color Cube](#)
2. [Itse1359-2245-Color-Explanation of the HSV Color Space](#)
3. [Itse1359-2250-Color-An Animated Dive Into an HSV Color Cylinder](#)
4. [Itse1359-2255-Color-Animated Fly Through an HSV Color Cylinder](#)

3. Skulpt

1. [Itse1359-2410-Getting Started with Skulpt](#)

4. Online Code Visualizer

1. [Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer](#)

Itse1359-1000-Preface

This module is the preface for a collection of modules designed for teaching ITSE 1359 Introduction to Scripting Languages: Python at Austin Community College in Austin, TX.

Table of contents

- [Welcome](#)
- [Information about the course](#)
 - [Course description](#)
 - [Prerequisites](#)
 - [Course rationale](#)
- [Learning resources](#)
 - [General](#)
 - [Primary learning resources](#)
- [How to use this instructional material](#)
- [An editorial comment](#)
- [Reinforcing what you are learning](#)
- [Miscellaneous](#)

Welcome

Welcome to the course material for *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

(Note to blind and visually impaired students: most of the material for this course is presented in plain text format and should be accessible using an audio screen reader or a braille display. While there are a few bitmapped images that aren't accessible, that material will not be needed for assignments or included on tests.)

Information about the course

Course description

As of September 2014, the description for this course reads as follows:

Introduction to scripting languages including basic data types, control structures, regular expressions, input/output, and textual analysis.

Prerequisites

One semester of programming or departmental approval.

Course rationale

This course is an introduction to scripting languages and Python. The purpose of the course is to prepare students for building scripts that control a sequence of program steps such as those used in developing testing and deploying software. A modern scripting language, Python, is used as an example of a scripting language.

Learning resources

General

I have long been a proponent of free and open educational resources for college students. This version of this course does not require students to purchase printed textbooks. Instead the learning resources for the course consist of a large number of online resources arranged in the **following major sections** :

1. Language Fundamentals - Demonstrate the basic techniques used to create scripts for automating system administrative tasks.

2. Testing - Design, code, and test Python applications using Python doctests and unit testing tools.
3. Text processing - Demonstrate the use of string operations and regular expressions in processing text.
4. Networking and Databases - Demonstrate the use of Python in developing applications using networking and databases.

Primary learning resources

You are probably reading this document online at cnx.org, otherwise known as **openstax cnx**. As of October 2014, two views of the document are available: an *openstax* view and a *legacy* view. The material is the same in both views. Only the format and the top-level organization is different.

In the *openstax* view, the group of documents is referred to as a **book** and each document is referred to as a **page**.

In the *legacy* view, the group of documents is referred to as a **collection** and each document is referred to as a **module**.

You should not think of this book or this collection as a comprehensive tutorial on Python programming. Instead, you should think of it as an *annotated guidebook* to free online resources that constitute the primary learning resources for the course.

In some cases, I will explain the material in detail and illustrate the material with working programs. In other cases, I will partially explain the material and refer you to one or more online resources for additional explanations. In some cases, I will simply refer you to one or more online resources for the complete explanation. In all cases, I will either attempt to explain the material or refer you to one or more online resources for a full or partial explanation.

For many of the modules, I will also provide a *review module* containing questions and answers to help you retain the important information provided by the module.

References to free online resources will include but will not be limited to the following websites listed in no particular order:

- [How To Think Like a Computer Scientist - Interactive Version](#)
- [How to Think Like a Computer Scientist](#)
- [Skulpt](#)
- [Online code visualizer](#)
- [learnpython.org](#)
- [Python at Codecademy](#)
- [Program Arcade Games With Python And Pygame](#)
- [tutorialspoint -- Python Tutorial](#)
- [The Python Tutorial](#)
- [A Beginner's Python Tutorial](#)
- [Another Beginner's Python Tutorial](#)
- [Diving Into Python](#)
- [Python/C API Reference Manual](#)
- [The Python Language Reference](#)
- [The Python Standard Library](#)
- [Hands-on Python Tutorial](#)
- [learnpython.org](#)
- [Non-Programmer's Tutorial for Python 3](#)
- [Python3 Tutorial](#)
- [Testing Your Code -- The Hitchhiker's Guide to Python](#)
- [TkDocs](#)
- [Hello Tkinter](#)
- [The Tkinter Grid Geometry Manager](#)
- [Python Course](#)
- [Tkinter 8.5 reference: a GUI for Python](#)
- [The Invent with Python Bookshelf](#) (*many online books, some free, some not free*)
- [Python Scientific Lecture Notes](#)

Several of the assignments require the use of Python Turtle Graphics. Here are links to some useful references on Turtle Graphics:

- [24.1. turtle - Turtle graphics - Python 3.4.3rc1 documentation](#)
- [Background Drawing Graphics - 50 Examples 1.0 documentation](#)

- [How to Think like a Computer Scientist Interactive Edition](#)
- [Notes on using Python's turtle built-in commands](#)
- [Simple drawing with turtle - Introduction to Programming with Python](#)

How to use this instructional material

Everyone is welcome to use this material in any manner that is consistent with the [Creative Commons](#) license under which it is published. However, if you are enrolled in Prof. Baldwin's section of this course at Austin Community College in Austin, TX, there are some other things that you need to know.

Although the URL may change in the future, for the Spring 2015 semester, you will find the online syllabus for the course at <http://www6.austincc.edu/directory/info.php?id=baldwin>.

Also, although the URL may change in the future, for the Spring 2015 semester, you will find the entry to the college website for the course at <http://www.austincc.edu/baldwin/>.

You will also have access to a [Blackboard](#) site that is specifically tailored to the course. You will find additional learning resources on that site including programming assignments and tests, assignment and test schedules, etc. The order of the programming assignments and tests mirrors the order of the topics covered in this Ebook. Therefore, you should study the material in the Ebook from start to finish in parallel with your efforts to complete the assignments and the tests.

An editorial comment

In the opinion of this author, the Python programming language in and of itself is not a particularly interesting programming language. In fact, to a programmer accustomed to compiled, strongly-typed programming languages such as C, C++, and Java, the Python programming language seems to be a little "sloppy" and fraught with pitfalls. However, in the grand scheme of things, as of 2015, Python is an **extremely important** programming language.

The importance of Python derives not from the language itself, but from the hundreds of independent open source Python libraries that have been developed by others in such areas as image manipulation, networking, plotting and graphics, engineering and scientific programming, web development, gaming, cryptography, database, geographic information systems (*GIS*) , audio and music, presentation, XML processing, etc. In fact, it is hard to come up with a programming application area where someone has not already supplemented the basic Python programming language with an open-source library designed for use in that application area. Many of those libraries are Python wrappers for compiled C code providing speed and efficiency not normally associated with interpreted languages such as Python. A long list of such library modules is provided at [UsefulModules](#). Many of the links on that page point to other pages that also contain lists of links. An even longer list is provided on the [SciPy Topical Software](#) page.

Reinforcing what you are learning

Although not required by the course, I highly recommend that you also study the material in the section of this Ebook titled [Putting Python to Work](#) in addition to the required material described [earlier](#). Seeing how Python is actually used in practice may help you to develop and retain the knowledge required to succeed in the course.

Similarly, I also recommend that you study the material on the following three websites in parallel with the material on this website to reinforce what you **are learning** :

1. [Python at Codecademy](#)
2. [Program Arcade Games With Python And Pygame](#)
3. [How To Think Like a Computer Scientist - Interactive Version](#)

The first website [listed above](#) provides an interactive online Python tutorial that will help you learn the fundamentals of Python programming and test your progress along the way.

The second website [listed above](#) will not only help you learn about Python programming, but may also provide you with some enjoyment along the way. This website provides both text and video instruction, along with quizzes and programming projects to teach you how to use the Python language to write interactive arcade games involving sound, graphics, etc. While much of that material is beyond the scope of this course, you may find that learning that material will make the course more enjoyable.

The third website [listed above](#) is an excellent free online interactive Python textbook designed specifically for use in courses commonly referred to by computer science academics as CS-1 courses. The non-interactive version of the book is located [here](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1000-Preface
- File: Itse1359-1000.htm
- Published: 10/13/2014
- Revised: 01/21/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1010-Getting Started

This module explains how to get started programming with Python.

Table of Contents

- [Preface](#)
 - [What you will learn](#)
 - [Beginners](#)
 - [Programmers](#)
 - [Python programmers](#)
 - [Windows would be handy.](#)
 - [Practice, practice, and more practice](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Why use Python?](#)
 - [What is Python?](#)
 - [More facts, less hype](#)
 - [Learn OOP without the details](#)
 - [A beginner-friendly language](#)
 - [Sneaking up on OOP](#)
 - [Python has an interactive mode](#)
 - [Python has a non-interactive mode](#)
 - [Python combines interactive and non-interactive modes](#)
 - [Small core, large library](#)
- [Let's Write a Program](#)
 - [Getting ready](#)
 - [Download and install the software](#)
 - [Starting the programming environment](#)

- [Enough talk -- let's begin](#)
- [What do you see?](#)
- [The python manuals](#)
- [The library reference](#)
- [IDLE \(Python GUI\)](#)
- [What does it look like?](#)
- [What is a GUI?](#)
- [The Python \(command line\) selection](#)
- [The command-line window](#)
- [Your first \(interactive\) Python program](#)
 - [Hello World in Python](#)
 - [The Python prompt >>>](#)
 - [Let's do it](#)
 - [Congratulations](#)
 - [Python is ready for more](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you will learn

There is something for just about everyone here. Beginners start at the beginning, and experienced programmers jump in further on. You simply need to enter the collection of modules at the point that best fits your prior programming knowledge.

Beginners

The official prerequisite for this course is "*One semester of programming or departmental approval.*" However, if you don't know anything about programming, these modules will help you to overcome that deficiency by taking you from ground zero to object-oriented computer programming using Python.

Programmers

If you already know how to program in some other language, these modules will teach you how to program using the Python scripting language.

Python programmers

If you already know how to program using Python, you can probably skip the modules in the "Language Fundamentals" section.

Windows would be handy

Although not absolutely necessary, it would be very handy for you to have access to, and know how to use Microsoft Windows. The programming skills that you will learn are applicable to a broad range of operating systems. However, I will use Microsoft Windows as my teaching platform, so you will need to be able to follow instructions explained in Windows jargon.

Practice, practice, and more practice

I am going to provide you with a lot of information, but you will also need to put your brain in gear and write a lot of programs to retain the information. Very few people become expert musicians without a lot of practice. Similarly, very few people become expert programmers without a lot of practice.

So, if you want to learn to program, plan to spend a lot of time in front of your computer, not just reading, but programming as well.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

(Note to blind and visually impaired students: the Figures in this module are bitmap images and are probably not accessible using an audio screen reader or a braille display. However, those Figures are simply screen shots of material that you can create on your own screen, which is probably accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Python 3.4.1 Shell
- [Figure 2](#). Python (command line) window.
- [Figure 3](#). Python Hello World.

Why use Python ?

With so many programming environments available, why use Python?

First of all, it's free. I like that. You won't need to invest an arm and a leg just to get started. All you will need to do is go to <http://www.python.org/> and download the software.

What is Python ?

You might also want to visit [The Python Tutorial](#), which is where I extracted the following:

Note: What is Python ?

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

More facts, less hype

So, beyond the hype, did I really choose Python because it is an interpreted, interactive, object-oriented programming language that is relatively easy to learn? The name of this course is *"ITSE 1359 - Introduction to Scripting Languages: Python"* so I didn't really have a choice.

Learn OOP without the details

With Python, you can learn some aspects of object-oriented programming without the need to understand many of the complex details. Note, however, that Python supports a very abbreviated form of OOP. You will have much more to learn about OOP when you move on to C#, C++, or Java.

A beginner-friendly language

Python is a beginner-friendly language that takes care of many of the complex details for you behind the scenes. This makes it possible for you to concentrate on the big picture of what your program is intended to accomplish without getting bogged down in detail.

Sneaking up on OOP

With Python, you can "sneak up" on object-oriented programming concepts. You don't have to deal with the object-oriented nature of the language on

the first day, or for a long time, for that matter. (*With Java, for example, you can't write and understand even the simplest program without encountering a need to understand some object-oriented programming concepts.*)

Python has an interactive mode

The interactive mode makes it easy to try new things without the *edit-compile-execute* cycle of other programming languages such as Java and C++. Interactive mode is a friendly environment for persons in the learning stages of a language.

Python has a non-interactive mode

Once you know that something works properly, it is very easy to *"put it in a can"* so to speak, and then execute it as a script. This lets you execute it without retyping it every time.

Python combines interactive and non-interactive modes

This combination gives you the ability to use previously written script files while working in the interactive mode. Using this approach, you are able to minimize your interactive typing effort while still writing and testing large Python programs in interactive mode.

Small core, large library

Like Java, Python has a small compact core and a large, extensible [library](#). Thus, much of what you will need to do has already been written and tested for you. Your task will be to write the code to glue those library components together, and to write new capabilities on an as-needed basis.

Let's Write a Program

Getting ready

Download and install the software

The first step is to go to <http://www.python.org/> with your web browser. Download, and install the Python software using the download link that you will find there.

I'm going to assume that either you already know how to do this, or you can get help from a friend. In other words, I'm not going to try to explain how to download and install the Python software.

Note: If you are working in an ACC computer lab on the Northridge campus, you don't need to download and install the Python software. It has already been installed. If you are working in an ACC computer lab on a different campus and you don't find the Python software, ask the lab manager to download and install it. If you are working on your computer at home, at some point you will need to download and install the Python software.

Starting the programming environment

As I mentioned earlier, all of my instructions will be based on the use of Python running on my Windows operating system.

However, as I also mentioned earlier, you have many choices for using Python. If you are running on some other platform, you will need to translate my instructions from Windows jargon into the jargon of your platform.

Enough talk -- let's begin

Find the *Start* button on the task bar on your Windows desktop. Select

Start/All Programs/Python x.y

where x.y is the version of Python that you have downloaded and installed. At the time of this writing, I have version 3.4.1 installed on my computer, and the selection shows on the menu as Python 3.4.

What do you see?

When you make this selection, you should see a menu having at least the **following five options** :

- IDLE (Python GUI)
- Module Docs
- Python (command line)
- Python Manuals
- Uninstall Python

The Python Manuals

Don't be bashful about selecting and reading the Module Docs and the Python Manuals. There is a wealth of information contained there, including a tutorial.

The library reference

A good way to get a feel for the breadth and power of Python is to select **Python Manuals** from the menu and then select the **Library Reference** link. (The [*Library Reference*](#) is also available online as of the time of this writing.)

Most of what you see there probably won't mean much to you at this point in time, but hopefully will be familiar territory after you complete this

course.

IDLE (Python GUI)

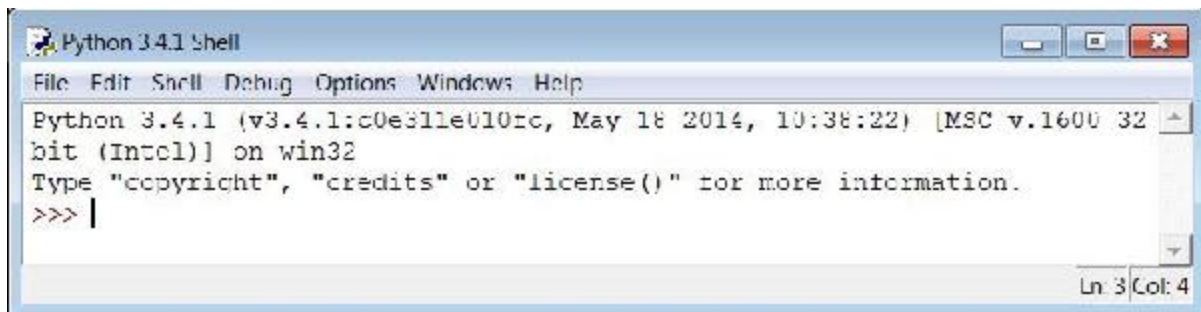
Selecting **IDLE (Python GUI)** from the menu brings up a window titled "*Python 3.4.1 Shell*" on my computer. This is one of the interactive programming environments available with Python on Microsoft Windows.

What does it look like ?

When you select **IDLE (Python GUI)** from the menu, you should see something like [Figure 1](#).

(Note to blind and visually impaired students: I don't know if you will find the Python Shell to be accessible. If not, don't worry. We will get to a command-line version that should be accessible a little later.)

Figure 1 . Python 3.4.1 Shell.



What is a GUI ?

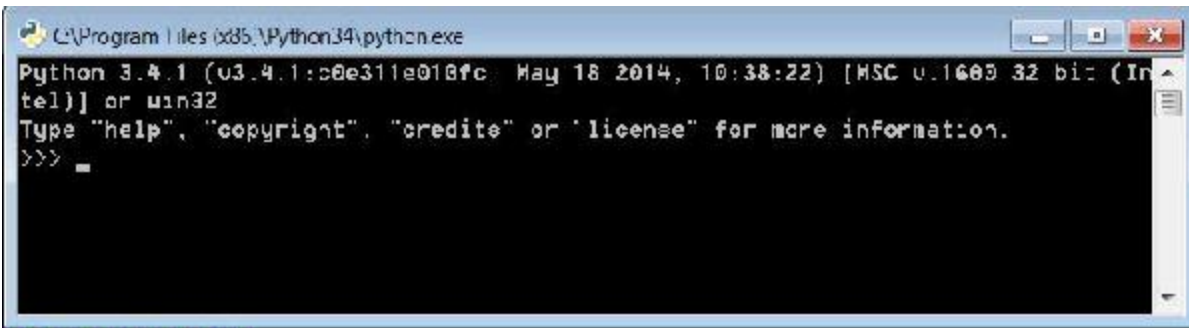
GUI is an acronym for *Graphical User Interface* . The IDLE window is a GUI. It can be used for interactive Python programming.

To make it go away when you are finished, simply click the button in the upper-right corner that is labeled with an X.

The Python (command line) selection

When I select the **Python (command line)** item in the menu mentioned [above](#), I get essentially the same thing as the GUI, but in a "black screen" window commonly referred to as a command-line window as shown in [Figure 2](#).

Figure 2 . Python (command line) window.



The command-line window

This window can also be used for interactive Python programming in much the same way that the Python Shell can be used. Each has some practical advantages and disadvantages.

To make this screen go away, click the X in the upper right-hand corner.

Note: In addition to IDLE, many Python development environments are available on the web - some free and some not free. My personal favorite is the free [Wing IDE](#). I have asked that it also be installed in the computer labs on the Northridge campus.

Your first (interactive) Python program

You can use IDLE, Wing (*if it is available*) , or the command line editor to write and execute your first Python program.

Hello World in Python

As has become the custom in programming circles, we will make our first Python program one that displays "Hello World" on the computer screen. We will write and execute it interactively.

The Python prompt >>>

The three right-angle brackets that you see in both of the interactive screens (>>>) make up the Python interactive prompt. When the cursor is blinking to the right of that prompt, you can enter a Python programming statement interactively.

Let's do it

Type the following text to the right of the Python prompt and press the Enter key:

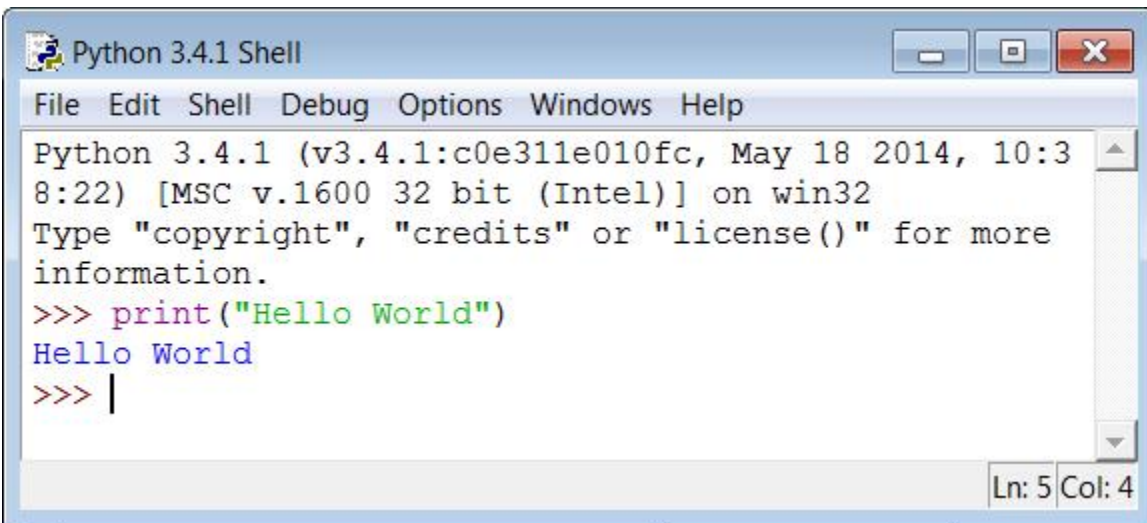
```
print("Hello World")
```

Note: The print function :

Prior to the release of version 3.0, print was a statement instead of a function and was executed without the parentheses.

If all goes well, your interactive Python screen should then look something like [Figure 3](#).

Figure 3 . Python Hello World.

A screenshot of a Python 3.4.1 Shell window. The window has a title bar that says "Python 3.4.1 Shell" and standard Windows window controls (minimize, maximize, close). Below the title bar is a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following text: "Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32", "Type 'copyright', 'credits' or 'license()' for more information.", ">>> print('Hello World')", "Hello World", and ">>> |". The text is color-coded: "print" is red, "Hello World" is green, and the prompts ">>>" are blue. A status bar at the bottom right shows "Ln: 5 Col: 4".

```
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
information.
>>> print('Hello World')
Hello World
>>> |
```

Pay particular attention to the line that reads **Hello World** following your entry. That is the output from your program.

Congratulations

You have just written (*and executed*) your first (*interactive*) Python program, and possibly your first computer program as well. Not only that, you only had to type one line of code to write and execute your program.

Note that your entire program, the output from your program, and a new prompt are all shown in [Figure 3](#).

Python is ready for more

Python has provided a new prompt so that you can expand your program, or write another one.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1010-Getting Started
- File: Itse1359-1010.htm
- Published: 10/13/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1010r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1010-Getting Started.

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1010-Getting Started* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? Python is useful only as a compiled language.

Go to [answer 1](#)

Question 2

True or False? Python is available for free.

Go to [answer 2](#)

Question 3

True or False? Python is an object-oriented language.

Go to [answer 3](#)

Question 4

True or False? Python has a large core and a small library.

Go to [answer 4](#)

Question 5

What does the Python interactive prompt look like?

Go to [answer 5](#)

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 5

The Python interactive prompt is >>>

Go back to [Question 5](#)

Answer 4

False. Just the opposite is true, small core and large library.

Go back to [Question 4](#)

Answer 3

True . However, because Python is not a strongly-typed language, some restrictions apply in comparison with the object-oriented nature of strongly-typed languages such as Java and C++.

Go back to [Question 3](#)

Answer 2

True. As of September 2014, Python can be downloaded for free from <https://www.python.org/>. The [license agreement](#) states in part:

"Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.4.1 alone or in any derivative version, provided, however, that ..."

Go back to [Question 2](#)

Answer 1

False. Python can be used in an interactive or interpreted mode. Although not discussed in the Getting Started module, there are also compilers available for Python.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1010r-Review
- File: Itse1359-1010r.htm
- Published: 10/13/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available

on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1015-Program Organization

This module provides a brief introduction to program organization using scripts, modules, and packages.

Table of contents

- [Preface](#)
- [Discussion](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

You are probably reading this document online at cnx.org, otherwise known as **openstax cnx**. As of October 2014, two views of the document are available: an *openstax* view and a *legacy* view. In the legacy view, the document is commonly referred to as a *module*, which is contained in a *collection* of modules. I mention this here to distinguish the word *module* from the usage of the same word that you will find [below](#).

This document provides a brief introduction to program organization using scripts, modules, and packages.

Discussion

Although you won't need this information initially, once your Python programs become larger than a few lines of code, you may **want to organize them into scripts, modules, and packages**. There is no shortage of online material on that topic. Here is a list of online resources that explain how to organize your programs:

- [tutorialspoint -- Python Modules](#)
- [The Python Tutorial -- 6. Modules](#)

- [Python Course -- Modular Programming and Modules](#)
- [The original sthurlow.com python tutorial -- Modules](#)
- [learnpython.org -- Modules and Packages](#)
- [A Beginner's Python Tutorial/Importing Modules](#)
- [Non-Programmer's Tutorial for Python 3/Using Modules](#)

Also, as you write more complicated programs, you will need to use modules that are contained in the standard Python distribution. Here is a link to an index of those modules in Python 3.4.2:

- [Python Module Index](#)

This information is being provided at this early stage in the course so that you will know where to find it when you need it.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1015-Program Organization
- File: Itse1359-1015.htm
- Published: 10/14/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1020-Numbers

This module explains various aspects of numbers in Python.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Let's program](#)
 - [Start the interactive Python environment](#)
 - [Program comments](#)
 - [How useful are comments?](#)
 - [A Python comment](#)
 - [Using the interactive mode](#)
 - [What is that ... prompt?](#)
 - [The sum of 2 and 5](#)
 - [Input and output](#)
 - [What was the input in this case?](#)
 - [What was the output?](#)
 - [Mixing comments and expressions](#)
 - [Command line versus GUI](#)
 - [Let's get technical](#)
 - [Statement ends at the end of the line](#)
 - [Literal values and variables](#)
 - [Operators](#)
 - [Operands](#)
 - [Unary and binary operators](#)

- [Some can be either](#)
- [Some arithmetic operators](#)
 - [Addition, subtraction, and multiplication](#)
 - [Division](#)
 - [Integer division](#)
 - [Do you remember long division?](#)
 - [The modulus operator](#)
- [The order of operations](#)
- [Operator precedence](#)
- [Grouping terms with parentheses](#)
 - [Forcing addition to be performed first](#)
 - [Forcing multiplication to be performed first](#)
 - [Nested parentheses](#)
- [Negative integer division](#)
- [Complex numbers](#)
- [Programming errors](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It explains various aspects of numbers in Python.

I will start out by showing you how to use Python as a programmable calculator. In the process, I will introduce you to some programming concepts, such as *operators* that I will explain in a more formal way in future modules.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

(Note to blind and visually impaired students: all of the Figures in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). IDLE GUI at startup.
- [Figure 2](#). A Python comment.
- [Figure 3](#). The sum of 2 and 5.
- [Figure 4](#). Mixing comments and expressions on the command line interface.
- [Figure 5](#). Example prompts from The Python Tutorial.
- [Figure 6](#). A Python addition operator.
- [Figure 7](#). Some arithmetic operators.
- [Figure 8](#). Two division operators.
- [Figure 9](#). Whole number quotient and remainder.
- [Figure 10](#). A simple expression.
- [Figure 11](#). A simple expression in Python.
- [Figure 12](#). Grouping terms with parentheses.
- [Figure 13](#). Nested parentheses.
- [Figure 14](#). Negative integer division.
- [Figure 15](#). An error on the command-line interface.
- [Figure 16](#). An error on the IDLE GUI interface.

Let's program

Start the interactive Python environment

The first thing that you need to do is to start the interactive programming environment. If you have forgotten how to do that, see the earlier module titled [Itse1359-1010-Getting Started](#) in [this book](#).

You can start the interactive Python environment from the Windows *Start* menu by selecting one of the following:

- IDLE (Python GUI)
- Python (command line).

Either way, the interactive programming environment should look something like [Figure 1](#) when it starts running.

Figure 1 . IDLE GUI at startup.

```
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18
2014, 10:38:22)
[MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for
more
information.
>>>
```

The >>> that you see on the last line is the Python interactive prompt, which I will refer to simply as the prompt.

Program comments

Before we go any further, I need to introduce you to the concept of *program comments* .

In programming jargon, a comment is text that you insert into the program that is intended for human consumption only. Comments provide a quick and easy form of documentation. They are ignored by the computer and are intended to explain what you are doing.

How useful are comments ?

Comments aren't terribly useful when doing interactive programming. Presumably you already know what you are doing and don't need to explain it to yourself using comments.

However, comments are very useful when you are writing scripts that you will store in files and use again later after you have forgotten how you did what you did.

In these modules, I will use comments occasionally to explain what I am doing for your benefit, even in interactive mode.

A Python comment

According to the [Python Language Reference -- 2.1.3 Comments](#) :

"A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax; they are not tokens."

[Figure 2](#) shows a Python comment, taken from the Python command-line interface, along with a new kind of prompt.

Figure 2 . A Python comment.

```
>>> # This is a comment  
...
```

Using the interactive mode

What is that ... prompt ?

The interactive mode actually uses two different kinds of prompts.

- One is >>>
- The other is ...

I will explain the difference a little later For now, just pretend like they both mean the same thing. That will suffice until we get into more complicated material.

The sum of 2 and 5

Enter 2+5 at the prompt in the IDLE GUI interface and press the Enter key. You should see something like [Figure 3](#) on your screen.

Figure 3 . The sum of 2 and 5.

Figure 3 . The sum of 2 and 5.

```
>>> # This is a comment in Python GUI shell
>>> 2+5
7
>>>
```

Input and output

We need to differentiate between input and output. Everything that appears on a line with one of the prompts is input to the Python interpreter. (*You typed it, so you know that it is input.*)

Everything that appears on a line without a prompt is output from the interpreter. (*You didn't type it. The interpreter produced it, so it was output.*)

What was the input in this case ?

Your input was the expression 2+5.

What was the output ?

Python evaluated that expression and produced an output, which was the sum of 2 and 5, or 7.

Then Python presented you with a new prompt to allow you to provide more input.

Mixing comments and expressions

Now using the Python command-line window, try entering the comment, followed by the expression that you see in [Figure 4](#), and note the difference in the prompts as compared to [Figure 3](#)..

Figure 4 . Mixing comments and expressions on the command line interface.

```
>>> # This is a comment in Python command-line  
interface  
... 2+5  
7  
>>>
```

My only reason for showing you this at this time is to give you some experience in mixing comments and expressions and to attempt to explain the difference between the >>> prompt and the ... prompt.

Command line versus GUI

The text in [Figure 4](#) was produced using the Python (command line) interface. I don't get exactly the same behavior (*with respect to prompts*) when I use the Python GUI interface. Rather, what I get is shown in [Figure 3](#). (*Note the missing ... prompt.*)

The difference between the ... prompt and the >>> prompt is discussed in section [2.1.2. Interactive Mode](#) of [The Python Tutorial](#). (*Note that the tutorial is also accessible in a different format from the Start/All Programs/Python 3.4 selection in Windows.*)

Apparently the command-line interface creates the ... prompt when it thinks that the next line is a continuation line. The example given in the tutorial is shown in [Figure 5](#). *(Don't be concerned if you don't understand the code in [Figure 5](#). I will explain similar code in a future module.)*

Figure 5 . Example prompts from The Python Tutorial.

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Apparently for the case of [Figure 4](#), the system became confused by the comment and treated the line after the comment as a continuation line.

The IDLE GUI did not become confused in [Figure 3](#). I believe that it accomplishes line continuation by automatically indenting the next line and no continuation was required in either [Figure 3](#) or [Figure 4](#).

Let's get technical

But not too technical. Computer programs are made up of statements.

Statements are composed of expressions.

(Later we will learn that large Python programs are made up of smaller programs called modules, which are made up of statements, which are composed of expressions.)

Statement ends at the end of the line

In Python, a statement normally ends at the end of the line that contains it (*although there are exceptions to this rule*) .

For the time being, suffice it to say that expressions are made up of

- literal values
- variables
- operators
- parentheses

Literal values and variables

Literal values are simply numbers that are included in an expression such as the 2 and the 5 shown in [Figure 3](#).

I will defer a discussion of variables until a future module, and will discuss operators and parentheses in the following paragraphs.

Operators

If you have ever used a hand calculator, you already know what operators are. The plus key on a hand calculator is an operator. The plus sign (+) in the expression shown in [Figure 6](#) is also an operator.

Figure 6 . A Python addition operator.

Figure 6 . A Python addition operator.

```
>>> 2+5
```

In programming jargon, operators are said to operate on *operands* ..

Operands

In [Figure 6](#), the 2 is the left operand and the 5 is the right operand of the + operator.

Unary and binary operators

Normally, operators are said to be either *unary* or *binary* .

A unary operator has only one operand while a binary operator has two operands.

Some can be either

Some operators, such as the minus sign (-), can be either unary or binary operators.

In its unary mode with a single operand, a minus sign is usually a *sign changing* operator, while in its binary mode with two operands, a minus sign is usually a subtraction operator.

All of the operators discussed in this module are being used as binary operators.

Some arithmetic operators

Python has numerous operators. You can find a complete list of operators in the [Python Language Reference](#).

For the time being, we will concentrate on the follow arithmetic operators:

- The addition operator, +
- The subtraction operator, -
- The multiplication operator, *
- The division operators, / and //
- The modulus operator, %

Addition, subtraction, and multiplication

[Figure 7](#) shows some examples of using these operators that probably won't present any surprises to you.

Figure 7 . Some arithmetic operators.

```
>>> 2+5
7
>>> 2-5
-3
>>> 2*5
10
>>>
```

If you add 2 and 5, you get 7. If you subtract 5 from 2, the answer is -3. If you multiply 2 by 5, you get 10

Division

Division is a bit more complicated. In its early days, Python had a single division operator (/) that behaved differently depending on the types of its operands. It behaved in a manner similar to the same division operator in Java and C++.

Somewhere along the way, a second division operator (//) was added to the language and the behavior of the original operator (/) was changed. As a result, the behavior is now quite a bit different from Java and C++. Take a look at [Figure 8](#).

Figure 8 . Two division operators.

```
>>> 2/5
0.4
>>> 2.0/5
0.4
>>> 2//5
0
>>> 2.0//5
0.0
>>>
```

Originally, *(in the early days of python)* the division operation shown on the first line in [Figure 8](#) would have yielded a value of 0. *(Note that both operands are integers with no decimal point.)*

The second division operation *(note the decimal point in the numerator)* would have yielded a value of 0.4 just like your hand calculator. The new division operator shown in the last two division operations did not exist.

Now the behavior of the division operator consisting of a single slash character (/) has been modified to produce a decimal fraction regardless of whether its operands are integers or are values containing a decimal point.

The new division operator consisting of two slash characters (//) produces the results shown in the last two division operations in [Figure 8](#).

Integer division

You probably won't see anything unusual in the second division operation in [Figure 8](#) and you may not see anything unusual in the first division operation in [Figure 8](#) *(unless you are an experienced Java or C++ programmer)* . However, the "integer division" shown in the third division operation in [Figure 8](#) probably merits an explanation.

In this case, we are dividing the integer 2 by the integer 5 producing an integer result.

Normally, a hand calculator would tell you that the answer is 0.4, but that is not an integer result. Rather, it is a decimal fraction.

As you can see in [Figure 8](#), Python tells you that the result of dividing the integer 2 by the integer 5 using the // division operator is 0.

Do you remember long division ?

Think back to when your second grade teacher taught you how to do long division with whole numbers. She told you that if you divide 2 by 5, you get

a quotient of 0 and a remainder of 5. Or, if you divide 23 by 4, you get a quotient of 5 and a remainder of 3. That is what we are talking about here. The third division operation in [Figure 8](#) produces a quotient of 0.

The modulus operator

And that brings us to the modulus operator (%).

Enter the code shown in [Figure 9](#). (You don't need to enter the comments. They are there to explain what is going on.)

Figure 9 . Whole number quotient and remainder.

```
>>> 23//4 #get integer quotient
5
>>> 23%4 #get remainder
3
>>> 23.0//4 #get quotient
5.0
>>> 23.0%4 #get remainder
3.0
>>>
```

It is probably safe to say that the purpose of the // division operator is to produce a whole number quotient (*no digits to the right of the decimal point even if the decimal point is showing as in the last operation in [Figure 8](#)*).

The purpose of the modulus operator is to produce the remainder resulting from a division.

As you can see from the example in [Figure 9](#), both division operations using the // division operator produced the whole number quotient of 5, and both modulus operations produced the whole number remainder of 3.

The order of operations

What is the result of evaluating the expression shown in [Figure 10](#) on a hand calculator?

Figure 10 . A simple expression.

$3+5*4$

Try it on your hand calculator. *(You will probably need to use an X instead of an * to indicate multiplication.)* My hand calculator gives an answer of 32.

Now try it with Python and you should get the result shown in [Figure 11](#).

Figure 11 . A simple expression in Python.

Figure 11 . A simple expression in Python.

```
>>> 3+5*4
23
>>>
```

Oops! This answer doesn't match the answer produced by my hand calculator, and I'll bet that it doesn't match your calculator either unless you are using a fancy scientific calculator.

The answer depends on the order in which you perform the various arithmetic operations. Ordinary hand calculators usually do the arithmetic in the order that the terms are fed into the keyboard.

Operator precedence

However, most computer programming systems, including Python, use a precedence system to decide which operations to perform first, which operations to perform second, etc.

I'm not going to go into the Python precedence system in detail. (*If you are interested in the order of precedence of all the operators, you can find a precedence table in [Python Language Reference -- 6.15. Operator precedence](#).*)

Rather, I am going to show you how to group terms using parentheses so that you can control the order of operations without worrying about the precedence system.

Grouping terms with parentheses

The Python code fragment in [Figure 12](#) shows how I can produce both results simply by grouping terms using parentheses.

Figure 12 . Grouping terms with parentheses.

```
>>> (3+5)*4 # do addition first
32
>>> 3+(5*4) # do multiplication first, default
23
>>>
```

The first expression produces 32 as produced by the hand calculator. The second expression produces 23 as produced by the earlier Python expression.

A Python expression is evaluated by first evaluating each of the sub-expressions inside the parentheses and then using those values to complete the evaluation.

Forcing addition to be performed first

In the first expression, the code in [Figure 12](#) forced Python to perform the addition first by placing the addition inside the parentheses. This produced an intermediate value of 8 when the sub-expression inside the parentheses was evaluated. The remaining part of the overall expression was then evaluated by multiplying the intermediate value by 4, producing a result of 32.

Forcing multiplication to be performed first

In the second expression, the code in [Figure 12](#) forced Python to perform the multiplication first (*which it does anyway by default, but the parentheses make that more obvious*). This produced an intermediate value of 20. The remaining part of the overall expression was then evaluated by adding the intermediate value to 3 producing an output of 23.

Hopefully, you get the picture. By using parentheses to group the terms in an expression, you have total control over the order in which the arithmetic operations are performed, without having to memorize a precedence table.

Nested parentheses

Parentheses can be, and often are nested to provide greater control over the order of the operations as shown in [Figure 13](#).

Figure 13 . Nested parentheses.

```
>>> (3+(5*4))*14 # nested parentheses
322
>>>
```

In this case, Python evaluates the expressions inside the innermost parentheses first, and then works from the inside out evaluating each pair of parentheses along the way.

Negative integer division

When I learned to do long division in the second grade, I didn't know about positive and negative numbers yet, so I didn't learn about remainders when one of the operands is negative and the other is positive. I suspect that you didn't either.

[Figure 14](#) shows how negative integer division and modulus works in Python.

Figure 14 . Negative integer division.

```
>>> 7 // -3
-3
>>> 7 % -3
-2
>>> 7 // 3
2
>>> 7 % 3
1
>>>
```

You may find these results surprising. I'm not going to try to explain it. I just want to make you aware that the behavior of integer division and integer modulus is different when the operands have different signs. I will leave it as "an exercise for the student" to think about this and come to a mental reconciliation with the facts as presented here.

Complex numbers

Python also provides a significant level of support for doing arithmetic with complex numbers. Since this is a very specialized area, which is probably of interest to only a small percentage of potential Python users, I'm not going to provide any of the details. If this is something that interests you, see [Standard Library -- Mathematical functions for complex numbers](#) for more information.

Programming errors

Sometimes, you may make an error and enter an expression that can't be evaluated. In this case, you will get an error message. A typical error message that was produced by the command-line interface is shown in [Figure 15](#).

Figure 15 . An error on the command-line interface.

```
>>> 3 + 5a
File "<stdin>", line 1
3 + 5a
    ^
SyntaxError: invalid syntax
>>>
```

Note that the IDLE GUI interface doesn't provide as much information as the command-line interface for the same error. Only the *SyntaxError* message appears on the IDLE GUI interface as shown in [Figure 16](#).

Figure 16 . An error on the IDLE GUI interface.

```
>>> 3 + 5a
SyntaxError: invalid syntax
>>>
```

Not shown here, however, is the fact that the **5a** is highlighted with a red background in the GUI interface. Also, the error message is displayed in red.

Referring back to [Figure 15](#), this error message means that the Python interpreter doesn't know how to add the value 3 to the value 5a. I will discuss error messages in more detail in a future module. I just wanted to show you a programming error here at the beginning. *(Note the ^ character in [Figure 15](#) pointing upward to the "a" character. This is a pointer to the source of the error.)*

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1020-Numbers
- File: Itse1359-1020.htm
- Published: 10/14/14
- Revised: 03/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1020r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1020-Numbers

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1020-Numbers* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? Python programming comments are ignored by the computer.

Go to [answer 1](#)

Question 2

True or False? Just like in C, C++, and Java, a Python comment begins with two slash characters (//) and continues to the end of the line.

Go to [answer 2](#)

Question 3

True or False? The only prompt used by the Python interactive system is the one consisting of three right-angle brackets (>>>).

Go to [answer 3](#)

Question 4

True or False? The output produced by the Python interactive system appears on a line without either of the prompts mentioned above.

Go to [answer 4](#)

Question 5

True or False? If you enter an expression at the prompt and press the Enter key, the result of evaluating the expression will be displayed on the next

line without a prompt.

Go to [answer 5](#)

Question 6

True or False? Computer programs are composed of expressions, which are made up of statements.

Go to [answer 6](#)

Question 7

In the following expression, $2+5$, what is the common jargon for the plus sign, and what is the common jargon for the 2 and the 5?

Go to [answer 7](#)

Question 8

List the operators discussed in the module titled *Itse1359-1020-Numbers* , and describe the purpose of each.

Go to [answer 8](#)

Question 9

True or False? Integer division produces a decimal result with non-zero digits to the right of the decimal point.

Go to [answer 9](#)

Question 10

True or False? The modulus operator is used to produce the quotient in division.

Go to [answer 10](#)

Question 11

Describe the use of parentheses in expressions.

Go to [answer 11](#)

Question 12

Describe how Python evaluates an expression containing parentheses.

Go to [answer 12](#)

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 12

A Python expression is evaluated by first evaluating each of the sub-expressions in the parentheses, and then using those values to complete the

evaluation. If the expression contains nested parentheses, the evaluation is performed by evaluating the innermost parentheses first and working outwards from there.

Go back to [Question 12](#)

Answer 11

Parentheses can be used to group terms in an expression in order to provide control over the order in which the operations are performed.

Go back to [Question 11](#)

Answer 10

False, the modulus operator is used to produce the remainder.

Go back to [Question 10](#)

Answer 9

False. Integer division produces an integer or whole number result with no non-zero digits to the right of the decimal point even if a decimal point is displayed.

Go back to [Question 9](#)

Answer 8

The operators and their purposes are:

- The addition operator, +

- The subtraction operator, -
- The multiplication operator, *
- The division operators, / and //
- The modulus operator, %

Go back to [Question 8](#)

Answer 7

The plus sign is commonly called the operator. The 2 and the 5 are commonly called operands. More specifically, the 2 is the left operand and the 5 is the right operand.

Go back to [Question 7](#)

Answer 6

False. Just the reverse is true. Programs are made up of statements, which are composed of expressions.

Go back to [Question 6](#)

Answer 5

True, unless the expression can't be evaluated, in which case an error message will appear.

Go back to [Question 5](#)

Answer 4

True.

Go back to [Question 4](#)

Answer 3

False. The Python interactive system also uses a secondary prompt consisting of three periods (...).

Go back to [Question 3](#)

Answer 2

False. In Python, a comment starts with the hash character (#) and ends at the end of the line.

Go back to [Question 2](#)

Answer 1

True. Programming comments are used for program documentation and are intended for human consumption.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1020r-Review
- File: Itse1359-1020r.htm

- Published: 10/14/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1030-Variables and Identifiers

This module provides an introduction to the use of variables, and the required syntax of the identifiers used to represent variables.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [What is a variable?](#)
 - [A pigeonhole in memory](#)
 - [The concept of type](#)
 - [Strongly typed languages](#)
 - [Python is not strongly typed](#)
 - [Declaration of variables](#)
 - [Dangerous curves ahead!](#)
 - [Don't use the same name for two variables](#)
 - [A more subtle danger](#)
- [Rules for identifiers](#)
 - [Give me the rules in plain English](#)
 - [Allowable characters](#)
 - [Case is significant](#)
 - [Numbers](#)
 - [Style](#)
- [Let's program](#)
 - [Start the interactive Python environment](#)

- [Create and use some variables](#)
 - [Back to the pigeonholes](#)
 - [The assignment operator](#)
 - [What is an operand?](#)
 - [Addition of variables](#)
 - [Assigning the same value to several variables](#)
 - [Create three variables](#)
 - [Assign different values](#)
- [A more comprehensive explanation](#)
 - [A code visualizer tool](#)
 - [Creation of variables and objects](#)
 - [The important point](#)
 - [The sum of two variables](#)
 - [Assigning the same value to several variables](#)
 - [Breaking a reference to an object](#)
- [Type considerations](#)
 - [Advantages and disadvantages](#)
 - [The range of values](#)
 - [Speed](#)
 - [Floating point provides greater range](#)
 - [Sometimes range is important, and sometimes not](#)
 - [Approximate results](#)
- [Automatic type handling in Python](#)
 - [Python takes care of routine type issues automatically](#)
 - [Assign some floating point values](#)
 - [How is this accomplished?](#)
- [The magic continuation variable](#)
 - [How does it work?](#)

- [Sum is saved in the continuation variable](#)
- [The primary purpose of the continuation variable](#)
- [Illegal variable names](#)
- [Variable name spelling errors](#)
 - [A serious programming problem](#)
 - [Why did this happen?](#)
 - [The code visualizer](#)
 - [Spelling errors can be dangerous](#)
 - [Defending against spelling errors](#)
 - [Meaningful variable names](#)
 - [Remember, case is significant in variable names](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module provides an introduction to the use of variables, and the required syntax of the identifiers used to represent variables.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

(Note to blind and visually impaired students: most of the Figures in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Create and use some variables.
- [Figure 2](#). Assigning the same value to several variables.
- [Figure 3](#). The truth about pigeonholes.
- [Figure 4](#). Assign a different value to the variable named b.
- [Figure 5](#). Python takes care of routine type issues automatically.
- [Figure 6](#). The magic continuation variable.
- [Figure 7](#). An illegal variable name.
- [Figure 8](#). A serious programming problem.
- [Figure 9](#). Visualizer showing result of a spelling error.

What is a variable?

As the name implies, a variable is something whose value changes over time.

A pigeonhole in memory

As a practical matter, a variable is a pigeonhole in memory, which has a nickname, where you can store values.

You can later retrieve the values that you have stored there by referring to the pigeonhole by its nickname (*identifier*). You can also store a different value in the pigeonhole later if you desire.

The concept of type

Strongly typed languages

One of the main differences between Python and programming languages such as Java and C++ is the concept of *type* .

In *strongly-typed* languages like Java and C++, variables not only have a name, they also have a type. The type determines the kind of data that you can store in the pigeonhole.

It is probably more correct to say that the type determines the values that you can store there and the operations (*addition, subtraction, etc.*) that you can perform on those values.

Python is not strongly typed

One of the characteristics that makes Python easier to use than Java is the fact that with Python you don't have to be concerned about the type of a variable. Python takes care of type issues for you behind the scenes. However that ease of use comes with some costs attached.

Declaration of variables

Another difference between Python and Java is that with Java, you must *declare* variables before you can use them. Declaration of variables is not required with Python.

With Python, if you need a variable, you simply come up with a name and start using it as a variable.

Dangerous curves ahead !

With this convenience comes some danger. You can only have one variable with the same name within the same scope (*I will discuss scope in a future module*) .

Don't use the same name for two variables

With Python, if you unintentionally use the same name for two or more variables, the first will be overwritten by the second. This can lead to program bugs that are difficult to find and fix.

A more subtle danger

A more subtle danger is that you create a variable that you intend to use more than once and you spell it incorrectly in one of those uses. This can be an extremely difficult problem to find and fix. I will illustrate what I mean by this later with a sample program.

Rules for identifiers

The name for a variable must follow the naming rules for identifiers that you will find in the [Python Language Reference -- 2.3. Identifiers and keywords](#).

Give me the rules in plain English

The notation used in the *Python Language Reference* to define the naming rules is a little complicated, so I will try to interpret it for you.

Allowable characters

I believe that the [Python Language Reference -- 2.3. Identifiers and keywords](#) is saying that identifiers must begin with either a letter or an underscore character. Following that, you can use an unlimited sequence of letters (*uppercase A through Z or lowercase a through z*), numbers (*0 through 9*), or underscore characters.

Note that although the underscore character is allowed, it has special meaning in Python. I recommend that you do not use the

underscore character for the identifiers that you create.

Case is significant

The letters can be uppercase or lowercase, and case is significant. In other words, the identifier **Ax** is not the same as the identifier **aX**.

Numbers

Numbers can be any of the digit characters between and including 0 and 9.

Style

Click [PEP 8 -- Style Guide for Python Code](#) to learn more about how to create identifiers for variables and other programming elements as well.

Let's program

Start the interactive Python environment

The first thing that you need to do is to start the interactive programming environment. If you have forgotten how to do that, see the module titled *Itse1359-1010-Getting Started*.

Create and use some variables

The interactive fragment shown in [Figure 1](#):

- Creates two variables named x and y,
- Populates them by assigning values of 6 and 5 to them respectively
- Adds their values together to produce the sum value of 11.

Figure 1 . Create and use some variables.

```
>>> x=6 # create and populate x
>>> y=5 # create and populate y
>>> x+y # add x to y and display the sum
11
>>>
```

Back to the pigeonholes

Using the informal jargon from an earlier paragraph, two pigeonholes are established in memory and are given nicknames of x and y.

The assignment operator

Integer values of 6 and 5 are stored in the two pigeonholes using the *assignment operator* (=).

The use of the assignment operator in this fashion causes the value of its *right operand* to be stored in the pigeonhole identified by its *left operand* .

What is an operand ?

If you don't recognize the use of the term operand, see the earlier module titled *Itse1359-1020-Numbers* for an explanation.

In this case, the right operands of the assignment operators are *literal numeric values* .

The left operands of the assignment operators are the nicknames identifying the two memory locations that constitute the variables named x and y.

Addition of variables

The values are retrieved from each pigeonhole and added together in the third line of code. The result of the addition (11) is displayed as output from the expression `x+y`.

Assigning the same value to several variables

Python allows you to assign the same value to several variables, causing them to come into existence (*begin to occupy memory*) at the same time if necessary

Consider the interactive fragment shown in [Figure 2](#).

Figure 2 . Assigning the same value to several variables.

```
>>> a=b=c=10 # assign 10 to several variables
>>> a+b+c     # add them together
30
>>> a=b=c=20 # assign 20 to same variables
>>> a+b+c     # add them together
60
>>>
```

Create three variables

The first line of code in [Figure 2](#) creates three variables named **a** , **b** , and **c** , and assigns a value of 10 to each of them.

They are then added together, in the second line of code, to produce an output value of 30.

Assign different values

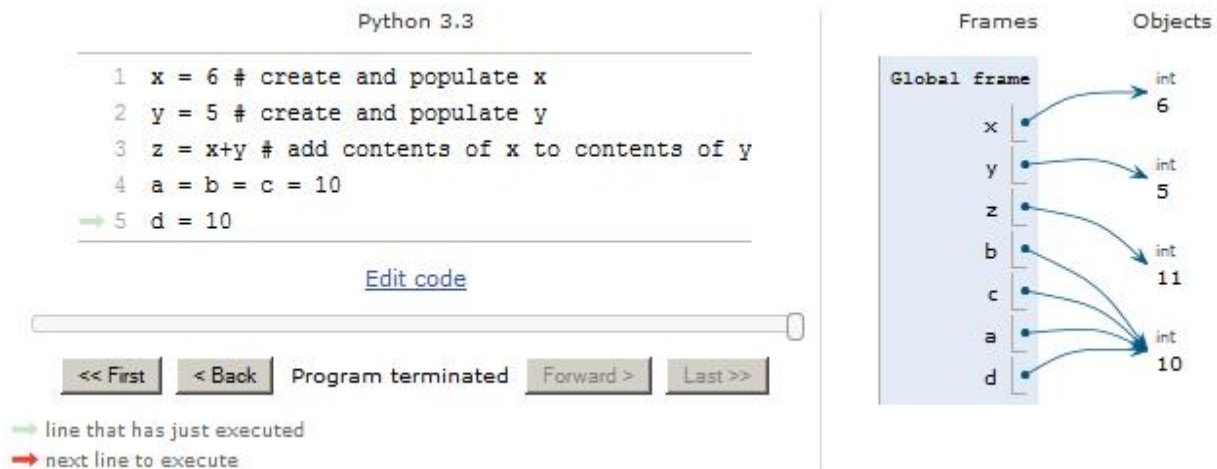
The fourth line of code assigns the value of 20 to the same three variables, replacing what was previously there with the new value. Again they are added together, this time producing an output value of 60.

A more comprehensive explanation

Pigeonhole explanations are good for starters, but let's move on to a more comprehensive and technically correct explanation.

According to the Python gurus, everything (*or at least almost everything*) in Python is an object. I indicated earlier that data values are stored in variables. In reality, that is not the case. [Figure 3](#) shows the truth about pigeonholes, variables, and objects.

Figure 3. The truth about pigeonholes.



A code visualizer tool

The image in [Figure 3](#) was produced using a *code visualizer tool* from the [Online Python Tutor](#). At this point in the course, you can ignore the buttons at the bottom of [Figure 3](#) and concentrate on the code window on the upper left and the diagram on the right.

Although not implemented using the Python interactive interface, the code shown in the code window in [Figure 3](#) is similar to code discussed earlier in this module.

Creation of variables and objects

The first two lines of code in [Figure 3](#) show the creation of two variables named `x` and `y` and the assignment of the values 6 and 5 to those variables respectively. The diagram on the right in [Figure 3](#) shows what actually happens in memory as those two lines of code are executed.

The two variables are created and stored in something called the *Global frame*. Two objects of type `int` are created and values of 6 and 5 are stored in those objects. The objects themselves are stored in a part of memory commonly called the *heap*.

The important point

Now here is the important point. The values of 6 and 5 are not actually stored in the variables. Instead, references (*sometimes called pointers*) that point to the objects containing the values of 6 and 5 are stored in the variables. Those references can later be used to find and to access the contents of the objects.

The sum of two variables

The statement on line 3 in [Figure 3](#) creates a new variable named **z** on the left side of the assignment operator. The expression on the right side of the assignment operator accesses the contents of the first two objects, adds their values, and assigns the result to the variable named **z**. This in turn causes a new object of type **int**

- to be created,
- populated with a value of 11 (*the sum of 6 and 5*), and
- stored on the heap.

A reference to the new object is stored in the new variable named **z**.

Assigning the same value to several variables

The statement on line 4 in [Figure 3](#) creates three more variables named **a**, **b**, and **c** and assigns a value of 10 to each of them. The diagram on the right indicates that a single new object of type **int** containing the value 10 is created and stored on the heap. References to that single object are stored in all three variables named **a**, **b**, and **c**. In other words, the contents of all three variables point to the same object.

Finally, on line 5, a value of 10 is stored in a new variable named **d**. The diagram on the right shows that a reference to the existing object containing the value 10 is stored in this variable as well. Thus, this variable shares an object with the variables named **a**, **b**, and **c**.

Breaking a reference to an object

[Figure 4](#) shows what happens if a different value is assigned to an existing variable. In this case, the string value "ok" is assigned to the existing variable named **b** by the last statement in the code in [Figure 4](#).

Figure 4. Assign a different value to the variable named b.



The diagram on the right in [Figure 4](#) shows that this causes a new object of type **str**

- to be created,
- populated with "ok", and
- stored on the heap.

It also causes a reference to the new object to be stored in the variable named **b**, replacing the reference to a different object that was previously stored there.

You will be learning a lot more about objects as well as the code visualizer tool in future modules.

Type considerations

In most modern programming systems, values having fractional parts, such as 3.14159 are commonly referred to as *floating point* types. (*This terminology comes from the fact that the decimal point can float back and forth from left to right.*)

Similarly, whole number values are commonly referred to as *integer* types. (*These are values with no decimal parts, such as, "I have 3 whole apples."*)

Advantages and disadvantages

Each type has advantages and disadvantages relative to the other when performing computations.

The range of values

For example, in some systems, the total range of values for a particular integer type is restricted to the set of whole numbers between -32768 and +32767. Anything outside that range cannot be handled as a whole number of that integer type.

Although the range of an integer type will be different on different systems, it will almost always be less than the range of a floating point type on the same system.

Speed

On some systems integer arithmetic is performed much faster than floating point arithmetic. On those systems, if speed is important, using integers may be more attractive than using floating point types.

Floating point provides greater range

On most systems, the floating point type provides a much greater range in terms of the values that can be maintained and used for arithmetic. For example, a particular system might be capable of representing the following two values as well as millions of values in between:

- 0.000000000033333
- 33333000000000.0

Sometimes range is important, and sometimes not

Sometimes range is important, and sometimes it isn't. However, as I mentioned above, in some systems this greater range is obtained at some sacrifice in arithmetic speed relative to integer types.

Approximate results

Also, as I will explain in the *Review* module that goes with this module, floating point arithmetic often produces approximate results instead of exact results.

While approximate results might be OK for some scientific calculations, they might not be OK for other calculations such as financial calculations for example.

Automatic type handling in Python

In *strongly-typed* languages such as Java, it is the responsibility of the programmer to make certain that types are handled correctly. For example, it is often not possible to store a floating point value into a variable previously declared to be for the storage of integer values. There is a very strong possibility that it simply won't fit.

Python takes care of routine type issues automatically

Consider the interactive code fragment shown in [Figure 5](#). The variables x and y are originally created to store integers and are populated with the values 5 and 6 respectively. The variables are added and the correct sum is displayed as output from the interpreter.

Figure 5 . Python takes care of routine type issues automatically.

```
>>> x=5
>>> y=6
>>> x+y
11
>>> x=5.55555
>>> y=6.66666
>>> x+y
12.22221
>>>
```

Assign some floating point values

Following that, the floating point values 5.55555 and 6.66666 are assigned to the same two variables named x and y. The two variables are successfully added and the correct result is displayed, demonstrating that the two floating point values were successfully stored in the variables originally created for integers.

How is this accomplished?

I don't know the inner workings of exactly how this is accomplished. As Python programmers, we don't usually care. We are simply happy that it works without the requirement for us to deal with the details of type.

The magic continuation variable

In interactive mode, Python automatically provides a variable whose name is simply the underscore character (`_`).

This variable makes it easy to do continuation arithmetic in interactive mode. *(This variable is intended for read only purposes, so don't assign a value to it explicitly.)*

At any point in time in interactive mode, this variable will contain the most recent output value displayed by the interpreter.

How does it work ?

Consider the interactive code fragment shown in [Figure 6](#). This fragment starts out just like previous examples, causing the sum of 5 and 6 to be calculated and displayed.

Figure 6 . The magic continuation variable.

Figure 6 . The magic continuation variable.

```
>>> 5+6
11
>>> _+22 # add 22 to the continuation variable
33
>>>
```

Sum is saved in the continuation variable

As mentioned above, the sum value of 11 is automatically saved in the continuation variable whose name is simply the underscore.

The contents of the continuation variable (*11*) are then added to 22 producing a result of 33. (*Note the use of the underscore as the left operand of the addition operator.*)

The primary purpose of the continuation variable

The primary purpose of this automatic variable named `_` is to make it easier for you to string calculations together in interactive mode and to display the intermediate results as you go.

Illegal variable names

The interactive fragment in [Figure 7](#) shows the result of attempting to use an illegal variable name.

Figure 7 . An illegal variable name.

```
>>> 1x=6
SyntaxError: invalid syntax
>>>
```

The output shown in [Figure 7](#) was produced by the IDLE GUI interface. Although not shown here, the "1x" was highlighted with a red background. *(The command-line interface provides essentially the same information but in a different format.)*

Variable names cannot begin with a digit. They must begin with either a letter or an underscore character. That was the reason for the error in [Figure 7](#). *(Without getting into the details as to why, I recommend that you never begin a variable name or a method name with the underscore character.)*

Variable name spelling errors

The interpreter assumes that you know what you are doing, and won't help you avoid spelling errors in variable names *(unless the spelling error produces an illegal variable name)* .

A serious programming problem

Now I will illustrate a very subtle and very serious programming problem. Consider the interactive code fragment in [Figure 8](#). The programmer expected to get a final answer of $16+5 = 21$, but instead the final answer was 11.

Figure 8 . A serious programming problem.

```
>>> xypdq = 6
>>> pzmbw = 5
>>> xypdq + pzmbw
11
>>> xyppq = 16 # accidental misspelling
>>> xypdq + pzmbw # correct spelling
11
>>>
```

Why did this happen ?

The problem arose in the fifth line of text in [Figure 8](#). In this line, the programmer intended to assign a value of 16 to the existing variable named xypdq. However, because of a spelling error, the programmer inadvertently created a new variable named xyppq and assigned the new value of 16 to the new variable instead of assigning it to the existing variable.

As a result, the value stored in the original variable named xypdq wasn't changed. When that variable was used later in an expression, the result did not meet the programmer's expectations.

The code visualizer

This error is illustrated in the diagram for the code visualizer in [Figure 9](#).

Figure 9. Visualizer showing result of a spelling error.



The code in the code window in [Figure 9](#) is similar to the interactive code in [Figure 8](#).

The diagram on the right in [Figure 9](#) shows a variable named **xyppq** (*not* *xypdq*) that exists solely because of a spelling error when writing the code. That variable should not exist. In addition, that variable points to an object of type **int** containing a value 16. That object also should not exist. (*The computer does exactly what you tell it to do, even if what you tell it to do is wrong.*)

Spelling errors can be dangerous

This is one of the greatest dangers of using a programming language that doesn't require the declaration of variables. This type of spelling error is easy to make (*as a result of a simple typing error*), and can be extremely difficult to find and fix.

Defending against spelling errors

The best defense against this kind of error is to make all of your variable names meaningful. Then if you make a typing error (*that results in a spelling error*), you might have a better chance of finding it later.

Meaningful variable names

Some meaningful variable names follow. Note the judicious use of upper and lower case to visually break the variable name into separate recognizable words. This is a naming convention that has become very popular, particular among Java programmers. It is commonly referred to as *camelCase* as in ***thisCamelHasFourHumps*** .

- myUpperLimit
- yourUpperLimit
- theOverheadRate
- theFinalPrice

Remember, case is significant in variable names

The variable named **MyUpperLimit** is not the same variable as the one named **myUpperLimit** .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1030-Variables and Identifiers
- File: Itse1359-1030.htm
- Published: 10/14/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1030r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1030-Variables and Identifiers.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1030-Variables and Identifiers* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? A variable is the same as a constant.

Go to [answer 1](#)

Question 2

True or False? Python is a *strongly typed* language.

Go to [answer 2](#)

Question 3

True or False? Python programmers must declare all variables.

Go to [answer 3](#)

Question 4

Explain the dangers of using a language that does not require variables to be declared.

Go to [answer 4](#)

Question 5

What is the best defense against spelling errors in variables names?

Go to [answer 5](#)

Question 6

True or False? Variable names can begin with numeric or digit characters.

Go to [answer 6](#)

Question 7

Write a simple program that illustrates case sensitivity in the names of variables.

Go to [answer 7](#)

Question 8

Explain the use of the assignment operator.

Go to [answer 8](#)

Question 9

Which type usually provides the greater range for storage of numeric values, integer or floating point?

Go to [answer 9](#)

Question 10

Should you just always use floating point instead of integer to be safe?

Go to [answer 10](#)

Question 11

Write a simple program that illustrates the approximation nature of floating point arithmetic.

Go to [answer 11](#)

Question 12

Explain the purpose of the automatic continuation variable whose name is simply the underscore character.

Go to [answer 12](#)

Figure index

- [Figure 1](#). Same letters, different case.
- [Figure 2](#). Same letters, same case.
- [Figure 3](#). Floating point errors.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 12

The primary purpose of the automatic variable named `_` is to make it easier for you to string calculations together in interactive mode and to display the

intermediate results as you go.

Go back to [Question 12](#)

Answer 11

See the sample program in [Figure 3](#). We know that the true result of this expression is an unending string of nines, as in 9.999999999999999

Figure 3 . Floating point errors.

```
>>> 10/3 + 20/3  
10.0
```

However, in this case, Python returned an answer of 10.0, indicating that the answer is accurate to three significant figures. This is not really the correct answer, but as a practical matter, it may be the *best* answer. I will leave that for you to decide.

Go back to [Question 11](#)

Answer 10

Probably not. Floating point arithmetic often suffers from speed penalties. In addition, integer arithmetic produces exact results while floating point

arithmetic usually produces approximate results (*although the approximations may be very close to being exact in many cases*) .

Go back to [Question 10](#)

Answer 9

Floating point usually provides the greater range for storage of numeric values.

Go back to [Question 9](#)

Answer 8

The assignment operator causes the value of its right operand to be stored in the memory location identified by its left operand.

Go back to [Question 8](#)

Answer 7

An example of such a program is shown in [Figure 1](#). Note that the names of the two variables have the same letters, but different case. The fact that the two variables are different variables is illustrated by the fact that each is assigned a different value. The sum of the two variables demonstrates that the two variables contain different, and correct, values.

Figure 1 . Same letters, different case.

Figure 1 . Same letters, different case.

```
>>> aX=10 # this is one variable
>>> Ax=20 # this is a different variable
>>> aX+Ax
30
>>>
```

Contrast the above result with the program in [Figure 2](#) where a variable whose name contains the same letters and the same case is used twice.

Figure 2 . Same letters, same case.

```
>>> ax=10
>>> ax=20
>>> ax+ax
40
>>>
```

All the program in [Figure 2](#) accomplishes is the assignment of two different values to the same variable. The second assignment overwrites the value previously assigned to the variable. The program then adds the variable to itself using its current value of 20 producing a result of 40 (*instead of 30 as in the previous example*) .

Go back to [Question 7](#)

Answer 6

False. Variable names must begin with a letter or underscore character.

Go back to [Question 6](#)

Answer 5

Use meaningful variable names for which the spelling is obvious, such as **theOverheadRate** .

Go back to [Question 5](#)

Answer 4

Several different kinds of problems can result from making typing errors that result in misspelling the names of variables. These errors usually result in programs that produce incorrect results without warning.

Go back to [Question 4](#)

Answer 3

False. Variable declarations are not required in Python. All that is required to cause a variable to come into existence is to invent a new name for a variable and assign a value to it.

Go back to [Question 3](#)

Answer 2

False. Python is not a strongly typed language. From a pure technical viewpoint, the Python programmer rarely needs to be concerned about type.

Go back to [Question 2](#)

Answer 1

False. The value of a variable is intended to change during the execution of the program. The value of a constant (*which I haven't discussed so far in this collection*) is not intended to change.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1030r-Review
- File: Itse1359-1030r.htm
- Published: 10/14/14
- Revised: 20/20/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1040-Strings Part 1

This module provides an introduction to the use of strings in Python.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Introduction to the string](#)
 - [Literals](#)
 - [An expression using variables](#)
 - [An expression using literals](#)
 - [String literals](#)
 - [A syntax error](#)
 - [A valid string literal](#)
 - [Proper syntax](#)
 - [Bad syntax](#)
 - [More examples](#)
 - [Triple-quoted strings](#)
 - [The newline \(\n\) character](#)
 - [Representing the newline character](#)
 - [An escape sequence](#)
 - [One more syntax option](#)
- [Escape sequences](#)
 - [The newline character](#)
 - [The built-in print function](#)

- [Including the newline character](#)
 - [The print function renders according to meaning](#)
- [The quote character](#)
 - [Escaping the quote character](#)
 - [Avoiding the quote problem](#)
- [List of escape sequences](#)
- [More ways to span lines](#)
 - [End the line with a backslash](#)
 - [Not restricted to strings](#)
 - [A form of concatenation](#)
 - [Use the \n escape sequence](#)
 - [Combine backslash and \n](#)
- [String concatenation](#)
 - [Literal string concatenation](#)
 - [Concatenate through placement](#)
 - [Creating whitespace](#)
 - [Using + for concatenation](#)
 - [Whitespace is included in the quotes](#)
 - [Visualizing string concatenation](#)
- [More on strings later](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin

Community College in Austin, TX. It provides an introduction to the use of strings in Python.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

(Note to blind and visually impaired students: most of the Figures in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). String literals.
- [Figure 2](#). Triple-quoted strings.
- [Figure 3](#). Triple-quoted strings with newlines.
- [Figure 4](#). Using the newline character.
- [Figure 5](#). Escaping the quote character.
- [Figure 6](#). Avoiding the quote problem.
- [Figure 7](#). End the line with a backslash.
- [Figure 8](#). Use the `\n` escape sequence.
- [Figure 9](#). Combine backslash and `\n`.
- [Figure 10](#). Concatenate through placement.
- [Figure 11](#). Creating whitespace.
- [Figure 12](#). Visualizing string concatenation.

Introduction to the string

The common interpretation of the word *string* in computer programming is that a string is a sequence of characters that is treated as a unit. For example, a person's first and last names are often treated as two different strings.

A person's first name usually consists of several characters and these characters are treated as a unit to produce a name.

Literals

Perhaps the best way to describe a literal is to describe what it is not.

A literal is not a variable. In other words, the value of a literal doesn't change with time as the program executes. You might say that it is taken at face value.

An expression using variables

For example, the following expression describes the sum of two variables named **var1** and **var2** :

sum = var1 + var2

The result of this expression can vary depending on the values stored in **var1** and **var2** at the instant in time that the expression is evaluated.

An expression using literals

On the other hand, the following expression describes the sum of two literal numeric values:

sum = 6 + 8

No matter when this expression is evaluated, it will always produce a sum of 14.

String literals

Literal values can also be used for strings. For example, the interactive code fragment in [Figure 1](#) shows

- My name entered three times, in three different ways, on the interactive command line.
- The output from the interpreter for each entry.

Figure 1 . String literals.

```
>>> "Dick Baldwin"
'Dick Baldwin'
>>> 'Dick Baldwin'
'Dick Baldwin'
>>> Dick Baldwin
SyntaxError: invalid syntax
>>>
```

The first two entries are valid string literals. As you can see, in the first two cases, the interpreter displays my name in the output.

Note that in the first two cases, my name is surrounded by either quotes (*sometimes called double quotes*) or apostrophes (*sometimes called single quotes*) .

A syntax error

However, the third entry is not a valid string literal, and the interactive interpreter produced a `SyntaxError` message. (Although not shown in [Figure](#)

[1](#), the IDLE GUI interface highlighted my last name with a red background immediately above the `SyntaxError` message.) In the third case, my name is not surrounded by either double quotes or single quotes, and that is what caused the error.

A valid string literal

Referring to *String and Bytes literals*, the [Python Language Reference](#) states:

Both types of literals can be enclosed in matching single quotes (') or double quotes (").

This explains why the first two input lines in the interactive code fragment in [Figure 1](#) were accepted and the third input line produced an error.

Proper syntax

In the first input line in [Figure 1](#), my name was surrounded by matching double quotes. In the second input line, my name was surrounded by matching single quotes.

Bad syntax

However, in the third input line, my name was not surrounded by quotes of either type and this produced a syntax error.

More examples

[Figure 2](#) shows two more examples of valid string literals.

Figure 2 . Triple-quoted strings.

```
>>> """Dick Baldwin"""
'Dick Baldwin'
>>> """Dick
Baldwin"""
'Dick\nBaldwin'
>>>

=====

>>> """Dick Baldwin"""
'Dick Baldwin'
>>> """Dick
... Baldwin"""
'Dick \nBaldwin'
>>>
```

The top half of [Figure 2](#) was produced by the IDLE GUI interface. The bottom half was independently produced by the Python command-line interface. I included both of them to show you the output format difference between the two interfaces. **In both cases, I pressed the Enter key** following the input that reads `"""Dick`.

Triple-quoted strings

Referring again to *String* and *Bytes* literals, the [earlier quotation](#) from the [Python Language Reference](#) goes on to state:

They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted*

strings). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

Further down that same page we find:

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A "quote" is the character used to open the string, i.e. either ' or ".)

When I pressed the Enter key as described [above](#), I entered an *unescaped newline* .

One of the main advantages of using triple-quoted strings is that this makes it possible to

- Deal with strings that occupy more than one line.
- Deal with all of the lines that make up the string as a unit.
- Preserve newline characters that separate the lines in the process.

This is illustrated in [Figure 3](#), which shows my name, surrounded by matching triple quotes and split onto two consecutive lines of input.

Figure 3 . Triple-quoted strings with newlines.

Figure 3 . Triple-quoted strings with newlines.

```
>>> """Dick
Baldwin"""
'Dick\nBaldwin'
>>>
```

The newline (\n) character

When this triple quoted, multiple-line input was displayed, by the interpreter, the display included "\n".

This is an "escape character" representation of the *newline* character. It appeared in the output at the point representing the end of the first line of input. This indicates that the interpreter knows and remembers that the input string was split across two lines.

Numerically, the newline character is represented by the following:

- 10 in decimal
- A in hexadecimal
- 012 in octal
- 00001010 in binary

Note: Historical note:

In case you are interested, very early versions of Python produced the following output for the input shown in [Figure 3](#):

```
'Dick\012Baldwin'
```

In those days, the newline character was represented by a backslash followed by its octal representation. (*If you don't know what octal means,*

don't worry about it. It was effectively superseded by hexadecimal about twenty years ago.)

Representing the newline character

As the name implies, a *newline* character is a character that means, "*Go to the beginning of the next line.*"

The newline character is sort of like the wind. You can't see the wind, but you can see the result of the wind blowing through a tree.

Similarly, you can't normally see a newline character, but you can see what it does. Therefore, we must represent it by something else, like `\n` if we want to be able to see where it appears within a string.

An escape sequence

The `\n` is what we call an *escape sequence*. I will discuss escape sequences in detail a little later in this module.

One more syntax option

The [Python Language Reference -- 2.4.1. String and Bytes literals](#) describes one more syntax option for strings as shown below. I am going to let this one lie for the time being. I will come back and address it in a future module if I have the time. I am including it here simply for completeness.

Both string and bytes literals may optionally be prefixed with a letter 'r' or 'R'; such strings are called raw strings and treat backslashes as literal characters. As a result, in string literals, `\U` and `\u` escapes in raw strings are not treated specially. Given that Python 2.x's raw unicode literals behave differently than Python 3.x's the 'ur' syntax is not supported.

Escape sequences

Escape sequences are special sequences of characters used to represent other characters that

- cannot be entered directly into a string, or
- would cause a problem if entered directly into a string.

The newline character

An example of a character that cannot be entered directly into a string is the newline character. Except when using triple quoted strings, you cannot enter the newline character directly into a string.

Why? Because when you press the *Enter* key in an attempt to enter a newline, that simply terminates your input for that line. It doesn't enter the newline character into the string.

The interactive code fragment in [Figure 4](#) illustrates the use of an escape sequence to enter the newline character into a string. Note the `\n` between my first and last names.

Figure 4 . Using the newline character.

Figure 4 . Using the newline character.

```
>>> print("Dick\nBaldwin")
Dick
Baldwin
>>>
```

The built-in print function

The Python interpreter has a number of functions built into it that are always available. They are listed in [The Python Standard Library -- 2. Built-in Functions](#) in alphabetical order. One of those functions is the **print** function.

The code in [Figure 4](#) calls the **print** function to display my name on two output lines.

Note: Historical note:

Up until version 3, Python had a **print** statement that did not require parentheses. In version 3, the **print** statement [has been replaced](#) with a **print()** function, with keyword arguments to replace most of the special syntax of the old **print** statement.

When the **print** function is used interactively, it is a request to have its argument (*the expression in parentheses*) printed on the next line. In [Figure 4](#), it is a request to have my name printed on the next line.

Including the newline character

In [Figure 4](#), I entered the newline escape sequence between my first and last names when I constructed the string. Then, when the string was printed, the cursor advanced to a new line following my first name and printed my last name on the new line. That is what escape sequences are all about.

The `print` function renders according to meaning

Note that the **`print`** function rendered the newline character according to its meaning.

By this, I mean that the **`print`** function did not print something that represented the newline character (`\n`) as we saw in [Figure 3](#). Instead, it actually did what a newline character is supposed to do -- go to the beginning of the next line.

The quote character

Escaping the quote character

Suppose that you are constructing a string that is surrounded by double quotes, and you want to use a pair of double quotes inside the string. If you were to simply enter the double quote when you construct the string, that quote would terminate the string.

The interactive code fragment in [Figure 5](#) shows how to escape the double quote character -- precede it with a backslash character.

Figure 5 . Escaping the quote character.

```
>>> print("Richard \"Dick\" Baldwin" )
Richard "Dick" Baldwin
>>>
```

What I mean by this is that if you want to include a double quote inside a string that is surrounded by double quotes, you must enter the double quote inside the string as shown in [Figure 5](#).

Avoiding the quote problem

Because this is such a common problem, and because the *escape solution* is so ugly and difficult to read, Python gives us another way to deal with quotes inside of quotes. This solution, shown in [Figure 6](#), is the use of single and double quotes in combination.

Figure 6 . Avoiding the quote problem.

```
>>> print('Richard "Dick" Baldwin')
Richard "Dick" Baldwin
>>>
```

In Python, double quotes can be included directly in strings that are surrounded by single quotes, and single quotes can be included directly in strings that are surrounded by double quotes. This is much easier to read than the solution that requires you to place a lot of backslash characters inside your string.

List of escape sequences

Escape sequences are a staple of modern computer programming and there is a lot of consistency from one language to the next. A list of the escape sequences that are honored by Python 3 is available in the [Python Language Reference -- 2.4.1. String and Bytes literals](#).

More ways to span lines

Just when you thought that you had seen it all, I am going to show you three more ways (*two new and one not so new*) to span multiple lines with strings. One of them is shown in [Figure 7](#).

End the line with a backslash

Figure 7 . End the line with a backslash.

Figure 7 . End the line with a backslash.

```
>>> print("Richard \  
Baldwin")  
Richard Baldwin  
>>>
```

As shown in [Figure 7](#), the use of a backslash at the end of the line makes it possible to continue the string on a new line. However, the backslash is not included in the output, and there is no newline character in the output.

Not restricted to strings

The backslash can be used at the end of a line to cause that line to be continued on the next line whether inside a string or not. This is illustrated in the review module following this module.

A form of concatenation

When used in this way with a string, the backslash at the end of the line becomes a form of string concatenation. The portions of the strings on each of the input lines are concatenated to produce a single line containing both parts of the string in the output.

I will have more to say about string concatenation later in this module.

Use the `\n` escape sequence

This isn't really new, but it so important that I decided to repeat it here for emphasis. As shown in [Figure 8](#), the inclusion of `"\n"` inside the string

caused the cursor to advance to a new line following my first name and printed my last name on the new line.

Figure 8 . Use the `\n` escape sequence.

```
>>> print("Richard \nBaldwin")
Richard
Baldwin
>>>
```

This is the common form of the newline escape sequence typically used in C, C++, and Java.

Combine backslash and `\n`

The code in [Figure 9](#) shows how to combine the backslash at the end of the line with a newline character placed there to cause the output to closely resemble the input.

Figure 9 . Combine backslash and `\n`.

Figure 9 . Combine backslash and \n.

```
>>> print("Richard \n\  
Baldwin")  
Richard  
Baldwin  
>>>
```

String concatenation

To concatenate two strings means to hook them together end-to-end, thus producing a new string that is the combination of the two.

Literal string concatenation

You can cause literal strings to be concatenated just by writing one adjacent to the other as shown in [Figure 10](#).

Concatenate through placement

Figure 10 . Concatenate through placement.

Figure 10 . Concatenate through placement.

```
>>> print("Dick" 'Baldwin')
DickBaldwin
>>> print('Joe'      "Smith")
JoeSmith
>>>
```

Note that you can mix the different quote types. Also, it doesn't matter if there is whitespace in between. The whitespace doesn't carry through to the output.

Creating whitespace

If you want any space between the substrings in the output, you must include that space inside the quotes that delimit the individual strings as shown in [Figure 11](#).

Figure 11 . Creating whitespace.

Figure 11 . Creating whitespace.

```
>>> x = "Richard "  
>>> y = " Baldwin"  
>>> print(x + "G." + y)  
Richard G. Baldwin  
>>>
```

Using + for concatenation

The plus operator (+) can be used to concatenate strings as also illustrated in [Figure 11](#).

This fragment assigns string literal values to two variables, and then uses the plus operator to concatenate the contents of those variables with another string literal.

Of course, it could also have been used to concatenate the contents of the two variables without the string literal in between.

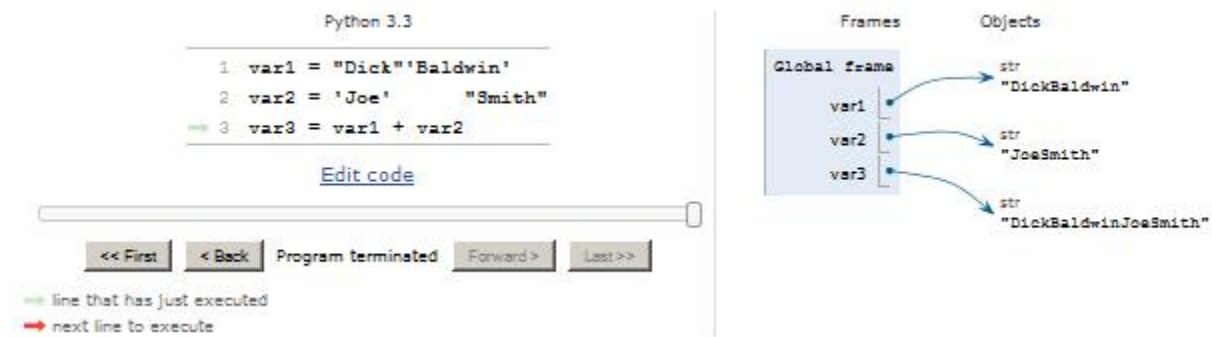
Whitespace is included in the quotes

Note that the string literals in [Figure 11](#) contain space characters. There is a space after the d in my first name and before the B in my last name. That is what I meant earlier when I said that if you want any space between the substrings in the output, you must include that space inside the quotes

Visualizing string concatenation

[Figure 12](#) illustrates string concatenation through the use of the [code visualizer](#) that you learned about in an earlier module.

Figure 12. Visualizing string concatenation.



The code in the code block in [Figure 12](#) is similar to the code in [Figure 10](#) and to the use of "+" operator for string concatenation. The diagram on the right shows how the concatenation of strings produces objects on the heap containing the concatenated strings.

The variable named **var1** points to an object of type **str** containing a string that was produced by concatenating two literal strings [through placement](#).

Similarly, the variable named **var2** points to a different object of type **str** containing a string that was produced by concatenating two different literal strings [through placement](#).

The variable named **var3** points to a third object of type **str** containing a string that was produced by using the "+" operator to concatenate the contents of two existing objects of type **str**. Although it might not be obvious at this point in the course, it is important to note that the contents of the third object contains the concatenation of copies of the contents of the first two objects. In particular, it doesn't simply contain pointers to the other two objects.

More on strings later

I will have more to say about strings in a future module. Before that, however, we need to learn how to create and execute script files, and we

also need to learn a little more about Python syntax.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1040-Strings Part 1
- File: Itse1359-1040.htm
- Published: 10/14/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1040r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1040-Strings Part 1.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1040-Strings Part 1* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

Describe the common meaning of the word *string* in your own words, and give some examples.

Go to [answer 1](#)

Question 2

Describe the common meaning of the word *literal* in your own words.

Go to [answer 2](#)

Question 3

Describe three different ways to format string literals (*without spanning lines*) and show examples.

Go to [answer 3](#)

Question 4

What is one of the advantages of using triple quoted strings? Show an example.

Go to [answer 4](#)

Question 5

Show two different representations of the *newline* character.

Go to [answer 5](#)

Question 6

Describe, in your own words, the purpose of an escape sequence. Show two examples.

Go to [answer 6](#)

Question 7

Show two different ways to include double quote characters in a string.

Go to [answer 7](#)

Question 8

Show the escape sequence for the tab character.

Go to [answer 8](#)

Figure index

- [Figure 1](#). Three ways to format string literals.
- [Figure 2](#). Triple-quoted strings.
- [Figure 3](#). Two representations of the newline character.
- [Figure 4](#). Example escape sequences.
- [Figure 5](#). Two ways to include double quote characters in a string.
- [Figure 6](#). The escape sequence for the tab character.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 8

The escape sequence for the tab character is `\t` as shown in [Figure 6](#).

Figure 6 . The escape sequence for the tab character.

```
>>> print("\tTom\n\tDick, and\n\tHarry")
      Tom
      Dick, and
      Harry
>>>
```

Go back to [Question 8](#)

Answer 7

Surround with single quotes, or use an escape character as shown in [Figure 5](#).

Figure 5 . Two ways to include double quote characters in a string.

Figure 5 . Two ways to include double quote characters in a string.

```
>>> print('Richard "Dick" Baldwin')
Richard "Dick" Baldwin
>>> print("Richard \"Dick\" Baldwin")
Richard "Dick" Baldwin
>>>
```

Go back to [Question 7](#)

Answer 6

Escape sequences are special sequences of characters used to represent other characters that either

- Cannot be entered directly into a string, or
- Would cause a problem if entered directly into a string.

Examples are shown in [Figure 4](#).

Figure 4 . Example escape sequences.

Figure 4 . Example escape sequences.

```
>>> print(\
    "She said, \"He \nwon't go\"")
She said, "He
won't go"
>>>
```

Go back to [Question 6](#)

Answer 5

Two representations are `\012` and `\n` as shown in [Figure 3](#). Of the two, the latter is probably the most commonly used.

Figure 3 . Two representations of the newline character.

```
>>> print('Richard\012G.\nBaldwin')
Richard
G.
Baldwin
>>>
```

Go back to [Question 5](#)

Answer 4

The use of triple-quoted strings, as shown in [Figure 2](#), makes it possible for you to continue a string on a new line, and to preserve the line break in the string.

Figure 2 . Triple-quoted strings.

```
>>> print("""Dick  
Baldwin""")  
Dick  
Baldwin  
>>>
```

Go back to [Question 4](#)

Answer 3

Surround the string with matching pairs of single quotes, double quotes, or triple quotes as shown in [Figure 1](#).

Figure 1 . Three ways to format string literals.

Figure 1 . Three ways to format string literals.

```
>>> print('Dick Baldwin')
Dick Baldwin
>>> print("Tom Jones")
Tom Jones
>>> print("""Mary Smith""")
Mary Smith
>>>
```

Go back to [Question 3](#)

Answer 2

Perhaps one way to describe the meaning of the word *literal* would be that the literal item is taken at face value, and its value is not subject to change as the program executes.

Go back to [Question 2](#)

Answer 1

The common interpretation of the word *string* in computer programming is that a string is a sequence of characters that is treated as a unit. For example, a person's first and last names are often treated as two different strings.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1040r-Review
- File: Itse1359-1040r.htm
- Published: 10/14/14
- Revised: 02/20/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1050-Introduction to Scripts

This module provides an introduction to the use of scripts in Python.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
 - [Important characteristics of scripts](#)
 - [Scripts are reusable](#)
 - [Scripts are editable](#)
 - [You will need a text editor](#)
 - [Combining scripts with interactive mode](#)
- [Getting started](#)
 - [The location of python.exe](#)
 - [Setting the path permanently](#)
 - [Setting the path temporarily](#)
 - [A Windows batch file](#)
 - [Setting the path with a batch file](#)
 - [Create a script file](#)
 - [Run your script file](#)
 - [The correct output](#)
 - [Alternative approaches](#)

- [Skulpt](#)
- [Online code visualizer](#)
- [How To Think Like a Computer Scientist](#)
- [Visualize your script file with the Online Python Tutor](#)
- [What's next?](#)
 - [Practice, practice](#)
 - [Cut-and-paste programming](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It provides an introduction to the use of scripts, and of necessity will depart from the interactive mode used in previous modules.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. However, there are a couple of bitmap images that may not be accessible.)

Figures

- [Figure 1](#). Screen output from running the script named 1359-1050-01.py.
- [Figure 2](#). Online Python Tutor visualizer screen output.
- [Figure 3](#). Visualizing a program from a future module.

Listings

- [Listing 1](#). Setting the path temporarily with a batch file.
- [Listing 2](#). Contents of the file named 1359-1050-01.py.

Introduction

This module provides an introduction to the use of scripts, and of necessity will depart from the interactive mode used in previous modules.

It will be necessary for me to get into a small amount of *system stuff* that really has nothing in particular to do with Python programming. Rather, it will involve getting your computer set up for using scripts with Python.

Up to this point in this collection, I have concentrated on the interactive programming capability of Python. This is a very useful capability that allows you to type in a program and to have it executed immediately in an interactive mode. But, interactive can be burdensome. By now you may have realized that you sometimes find yourself typing the same thing over and over. That is where scripts are useful.

Important characteristics of scripts

Scripts are reusable

Basically, a script is a text file containing the statements that comprise a Python program. Once you have created the script, you can execute it over and over without having to retype it each time.

Scripts are editable

Perhaps, more importantly, you can make different versions of the script by modifying the statements from one file to the next using a text editor. Then you can execute each of the individual versions. In this way, it is easy to create different programs with a minimum amount of typing.

You will need a text editor

Just about any text editor will suffice for creating Python script files and you may already have a favorite. Whichever editor you choose, make certain that it produces plain ASCII text in its output (*no bold, no underline, no italics, etc.*) .

In addition to functioning as an interactive interface, the IDLE GUI interface also provides a text editor for creating scripts. Once you have the GUI interface open, select **New File** from the **File** menu. After that, the behavior is similar to many other text editors.

Combining scripts with interactive mode

It is also possible to combine script files with interactive mode to incorporate pre-written scripts into interactive programs.

Getting started

In order to use script files, you must prepare your computer to use them or you must pursue one of the [alternative approaches](#) described later. Preparing your computer for use with Python doesn't amount to much in the way of effort, but it is critical. All of the following instructions are based on the Windows operating system. If you are using some other operating system, you will need to translate the instructions for use with that operating system.

The location of python.exe

When you first installed your Python software, a directory should have been created somewhere on your hard drive containing a file named **python.exe** . You will need to locate that file. On my Windows machine, the file is located in the following folder:

C:\Program Files (x86)\Python34

Setting the path permanently

In case you are unfamiliar with the term, the "*path*" is a text string, otherwise known as an *environment variable* , that tells the operating system where to search for executable files such as the file named **python.exe** .

You will need to cause the directory containing the file named **python.exe** to be listed in your *system environment variable* named *path* , either permanently or temporarily.

If you already know how to set the path, and you want to make a permanent change to the path, go ahead and do it. If you don't already know how, you may need to get some help.

I'm not going to try to tell you how to make a permanent change to the path. If you don't do it correctly, you may cause problems that are difficult to recover from. I don't want to be responsible for that. However, I will tell you how to make a temporary change to the path in the next section.

Setting the path temporarily

A Windows batch file

A Windows batch file is an executable text file with an extension of .bat. In its simplest form, it contains one or more commands that you would otherwise enter, one at a time, at the command-line prompt. When you execute the batch file, it attempts to cause those commands to be executed sequentially and automatically.

I am a big believer in the use of batch files for a variety of reasons, not the least of which is the ability to type a set of commands once and to execute them many times without having to retype them.

Batch files also make it possible to execute commands on the command line in the "current directory" without the requirement to go through the steps that are normally required to open a command-line window and navigate to the current directory.

Setting the path with a batch file

[Listing 1](#) shows the contents of a batch file named **1359-1050-01.bat** containing four commands that :

1. Reduce the clutter on the output screen (*echo off*) .
2. Temporarily set the path to the location of the file named **python.exe** on my computer.
3. Attempt to execute the Python script file named 1359-1050-01.py, which will be discussed later in this module.
4. Pause onscreen and wait to be dismissed with the prompt "*Press any key to continue . . .*"

Listing 1 . Setting the path temporarily with a batch file.

Listing 1 . Setting the path temporarily with a batch file.

```
echo off

path=%path%;"C:\Program Files (x86)\Python34"

python 1359-1050-01.py

pause
```

The second command in [Listing 1](#) is the important command for this section of this module. It will temporarily set the path to the indicated folder. If you elect to use this approach, you will need to replace the text inside the double quotes with the path to the folder containing the file named **python.exe** on your computer.

This approach appends the new path onto the environment variable named *path* on a temporary basis. It goes away as soon as you respond to the **pause** statement at the end or otherwise close the command-line window that appears when you execute the batch file.

Create a script file

Once you have the path variable properly set, or you know how to set it temporarily in a batch file, use the text editor of your choice and create a file named **1359-1050-01.py** that contains the Python programming statements shown in [Listing 2](#). *(You can give the file any name that you choose provided that the extension is .py and provided that you properly reflect the name in the third line of the batch file shown in [Listing 1](#).)*

Listing 2 . Contents of the file named 1359-1050-01.py.

```
a=2
b=3
a=2*a
b=3*b
c=a+b
print(c)
```

You should recognize all of the statements in [Listing 2](#). Note, however, that in a script, they are not preceded by an interactive prompt (>>>).

Store this file in any folder on your hard drive. You may want to create a new directory for the sole purpose of storing Python script files.

Then create the corresponding batch file with the contents shown in [Listing 1](#) and save it in the same folder as the script file. *(If you have permanently set your path to support Python, you can omit the path command in [Listing 1](#).)*

Run your script file

Once you have accomplished the above, double-click the batch file to execute it. The batch file will temporarily set the path and attempt to run your script. If all goes well, a command-line window will appear on your screen with contents similar to that shown in [Figure 1](#). *(Note that I deleted some of the text on the first line for brevity. Your first line will be different anyway.)*

Figure 1 . Screen output from running the script named 1359-1050-01.py.

```
M:\Baldwin\AA-School\...>echo off
13
Press any key to continue . . .
```

The correct output

If you go back to the script file in [Listing 2](#) and do the arithmetic, you will see that the output value of 13 produced by the program is correct.

If you got an output value of 13, -- congratulations -- you have just written and executed your first Python script. If not, try to figure out why.

Alternative approaches

There are at least three (*and probably more*) alternative online approaches that allow you to write and run Python code without a requirement to install Python on your computer:

- [Skulpt](#)
- [Online code visualizer](#)
- [How To Think Like a Computer Scientist - Interactive](#)

All three of these approaches run in a browser and can even be used with a tablet or a smart phone.

Skulpt

[Skulpt](#) is an entirely in-browser implementation of Python. No local installation, plugins, or server-side support is required.

Just go to www.skulpt.org in your computer, tablet, or smart phone, scroll down to the **Demo** pane, type in your Python code and click the **Run** button below the **Demo** pane. The program output will appear in the **Output** pane below the **Run** button.

You can learn more about using Skulpt, including how to embed Skulpt in your HTML pages in the module titled [Itse1359-2410-Getting Started with Skulpt](#) in [this book](#).

Online code visualizer

See the section titled [Visualize your script file with the Online Python Tutor](#) later in this module for a discussion of an online code visualizer. This tool also runs in a browser and can be used with a computer, a tablet, or a smart phone.

How To Think Like a Computer Scientist

[How To Think Like a Computer Scientist - Interactive](#) is an online interactive textbook that can be freely accessed in your browser on a computer, a tablet, or a smart phone. It uses the two tools described above plus video, self-check quizzes, and other features to teach Python programming. There are numerous opportunities within the book for you to write and run Python code completely within your browser.

(Note that as of February, 2015, the online video in the book seems to have an audio compatibility problem with my Android tablet. I have discussed this with the author and I expect the problem to be corrected soon.)

Visualize your script file with the Online Python Tutor

When learning to program, it is often useful to be able to follow the execution of a program step-by-step. A free online tool called the [Online Python Tutor](#) makes this possible. (You have seen some output from this tool in earlier modules. I will provide a more detailed explanation of how to use the tool in the module titled [Itse1359-1065-Visualizing Python](#) in [this book](#).)

The following quotation was copied from [pythontutor.com](#).

*" [Online Python Tutor](#) is a free educational tool created by [Philip Guo](#) that helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this tool, a teacher or student can write **Python**, **Java**, and **JavaScript** programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs."*

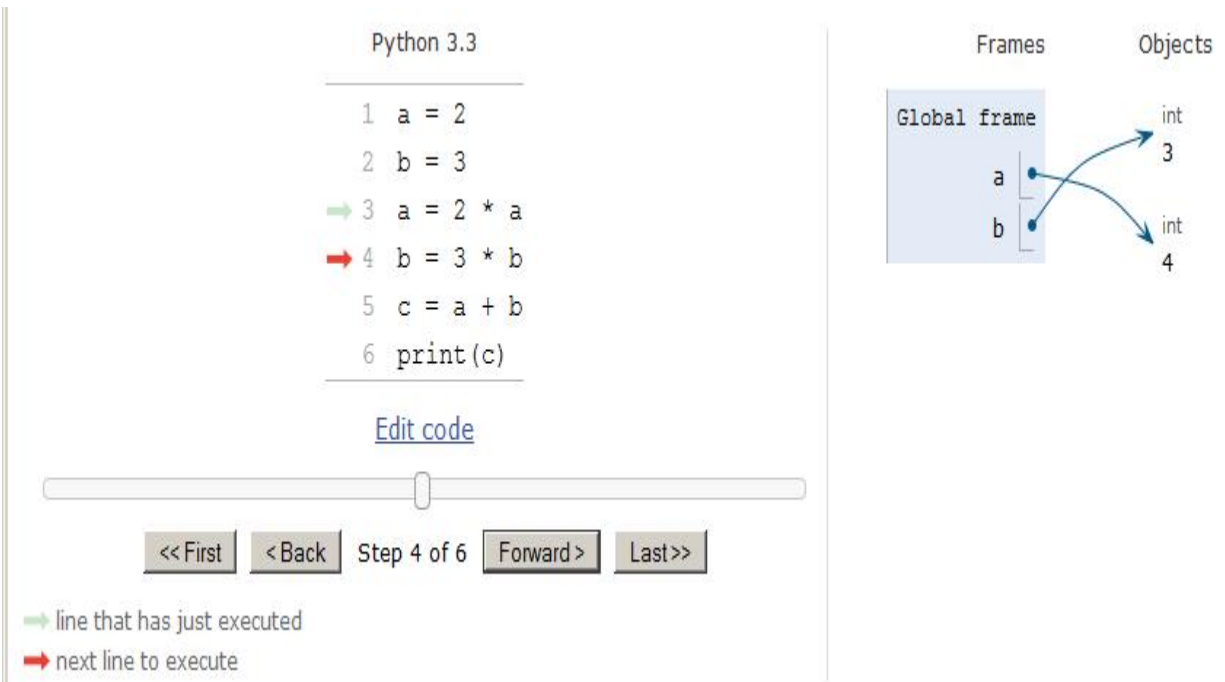
Beginning with this module and continuing with future modules, I will recommend that you

- Open the [Online Python Tutor](#).
- Copy the example programs, such as the code shown in [Listing 2](#), into the online [text editor](#).
- Select Python 3.3 in the pull-down list.
- Click the button labeled **Visualize Execution**.

The screen will then change to include what you see in [Figure 2](#). Use the **Forward** and **Back** buttons to step forward and backward through your code, one instruction at a time, while observing the information that appears on the screen to the right of the code window.

For example, [Figure 2](#) shows the [Online Python Tutor](#) visualizer screen after the first three statements from [Listing 2](#) have been executed. The information on the right shows the values stored in the variables named **a** and **b** at that point in the execution of the program.

Figure 2 . Online Python Tutor visualizer screen output.



As you might expect, more complicated code produces more complicated output from the visualizer. For example, [Figure 3](#) shows the output from the [Online Python Tutor](#) when visualizing a program from a future module titled [Itse1359-1440-Class Variables](#) in [this book](#).

Figure 3 . Visualizing a program from a future module.

Python 3.3

```
1 class TestClass(object):
2     def addClassVar(self,data):
3         TestClass.classVar02 = data
4
5 print("Instantiate and display two objects of TestClass")
6 ref01 = TestClass()
7 print(ref01)
8 ref02 = TestClass()
9 print(ref02)
10
11 print("Add a class variable directly.")
12 TestClass.classVar01 = "ABCD"
13
14 print("Add a class variable via an object.")
15 ref01.addClassVar("1234")
16
17 print("Display both class variables via one object")
```

[Edit code](#)

<< First < Back Program terminated Forward > Last >>

→ line that has just executed
→ next line to execute

Program output:

```
Instantiate and display two objects of TestClass
<_main_.TestClass object at 0x7f08a8d07b90>
<_main_.TestClass object at 0x7f08a8d046d0>
Add a class variable directly.
Add a class variable via an object.
Display both class variables via one object
ref01: ABCD
ref01: 1234
Display both class variables via the other object
ref02: ABCD
ref02: 1234
|
```

Frames

Objects

Global frame

TestClass

ref01

ref02

addClassVar

self

data

Return value

NoneType

None

str

"addClassVar"

function

addClassVar(self, data)

TestClass class

hide attributes

TestClass empty instance

TestClass empty instance

str

"classVar01"

str

"ABCD"

str

"1234"

str

"classVar02"

What's next ?

Obviously, there is a lot more that you will need to learn before you can write that "killer app" that takes the world by storm, but at this point, you have the tools to experiment with some simple scripts.

Practice, practice

I recommend that you look back into the earlier modules and convert some of the interactive programs listed there into scripts, execute them, and confirm that the scripts behave as expected.

Cut-and-paste programming

In future modules, I will switch back and forth between scripts and interactive mode, depending on which seems to be the most appropriate at the time.

I am a strong advocate of cut-and-paste programming. Cut-and-paste programming works well with scripts, and not so well with interactive mode.

Therefore, any time there is very much typing involved, you can usually expect to see me using scripts instead of interactive mode.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1050-Introduction to Scripts
- File: Itse1359-1050.htm
- Published: 10/14/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1050r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1050-Introduction to Scripts.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1050-Introduction to Scripts* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

In your own words, what is a script?

Go to [answer 1](#)

Question 2

True or False? A script can be used only once and must then be discarded.

Go to [answer 2](#)

Question 3

Convert the interactive program shown in [Figure 1](#) into a script, execute it, and confirm that you get the correct result.

In case you didn't get any output, note that unlike in interactive mode, in order to cause a script to produce an output, you will need to use a statement something like the following:

```
print(x+y)
```

Figure 1 . Question 3.

Figure 1 . Question 3.

```
>>> x=6 # create and populate x
>>> y=5 # create and populate y
>>> x+y # add x to y and display the sum
11
>>>
```

Go to [answer 3](#)

Question 4

Convert the interactive program shown in [Figure 3](#) into a script and execute it.

Figure 3 . Question 4.

```
>>> a=b=c=10 # assign 10 to several variables
>>> a+b+c # add them together
30
>>> a=b=c=20 # assign 20 to same variables
>>> a+b+c # add them together
60
>>>
```

Go to [answer 4](#)

Question 5

Convert the interactive program shown in [Figure 5](#) into a script and execute it.

Figure 5 . Question 5.

```
>>> x=5
>>> y=6
>>> x+y
11
>>> x=5.55555
>>> y=6.66666
>>> x+y
12.22221
>>>
```


Go to [answer 5](#)

Question 6

Convert the interactive program shown in [Figure 7](#) into a script and execute it.

Figure 7 . Question 6.

```
>>> 5+6
11
>>> _+22 # add 22 to the continuation variable
33
>>>
```



Go to [answer 6](#)

Question 7

Convert the interactive program shown in [Figure 9](#) into a script and execute it.

Figure 9 . Question 7.

```
>>> print("Dick\nBaldwin")
Dick
Baldwin
>>>
```

Go to [answer 7](#)

Figure index

- [Figure 1](#). Question 3.
- [Figure 2](#). Answer 3.
- [Figure 3](#). Question 4.
- [Figure 4](#). Answer 4.
- [Figure 5](#). Question 5.
- [Figure 6](#). Answer 5.
- [Figure 7](#). Question 6.
- [Figure 8](#). Answer 6.
- [Figure 9](#). Question 7.
- [Figure 10](#). Answer 7.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 7

One possible solution is provided in [Figure 10](#).

Figure 10 . Answer 7.

```
print("Dick\nBaldwin")
```

Go back to [Question 7](#)

Answer 6

One possible solution is provided in [Figure 8](#).

Figure 8 . Answer 6.

```
x = 5+6  
print(x)  
x = x + 22  
print(x)
```

If you were unable to get this to work, don't be dismayed. The special variable whose name is the underscore character is available only in interactive mode. Therefore, you can't use that variable in a script, and you will need to develop a workaround.

Go back to [Question 6](#)

Answer 5

One possible solution is provided in [Figure 6](#).

Figure 6 . Answer 5

Figure 6 . Answer 5

```
x=5
y=6
print(x+y)
x=5.55555
y=6.66666
print(x+y)
```

Go back to [Question 5](#)

Answer 4

One possible solution is provided in [Figure 4](#).

Figure 4 . Answer 4.

```
a=b=c=10 # assign 10 to several variables
print(a+b+c) # add them together
a=b=c=20 # assign 20 to same variables
print(a+b+c) # add them together
```

Go back to [Question 4](#)

Answer 3

One possible solution is provided in [Figure 2](#).

Figure 2 . Answer 3.

```
x=6 # create and populate x
y=5 # create and populate y
print(x+y) # add x to y and display the sum
```

Go back to [Question 3](#)

Answer 2

False. Script files are reusable.

Go back to [Question 2](#)

Answer 1

A script is a text file containing the programming statements that comprise a Python program.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1050r-Review
- File: Itse1359-1050r.htm
- Published: 10/14/14
- Revised: 02/21/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1060-Syntax

This module will help you to understand the physical construction of a Python program with particular emphasis on indentation.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
- [Program construction](#)
 - [A physical line](#)
 - [A logical line](#)
 - [Statements and logical lines](#)
 - [Comments](#)
 - [Explicit line joining](#)
 - [The backslash character](#)
 - [For illustration only](#)
 - [No spaces or comments allowed](#)
 - [Otherwise illegal](#)
 - [Implicit line joining](#)
 - [Blank lines](#)
- [Indentation](#)
 - [Indentation is used to determine grouping of statements](#)
 - [A blessing and a curse](#)
 - [No safety nets](#)
 - [An example of correct indentation](#)
 - [Compare A with B and take appropriate action](#)
 - [A compound statement](#)
 - [Group behavior](#)
 - [Another sample script](#)
 - [Members of the group](#)
 - [An opinion](#)
 - [Indentation details](#)

- [Leading whitespace](#)
- [Tabs](#)
- [The bottom line on indentation](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It explains some of the mechanics of putting together a Python program.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Two physical lines.
- [Figure 2](#). Output from the script in Listing 4.

Listings

- [Listing 1](#). Explicit line joining.
- [Listing 2](#). Implicit line joining.
- [Listing 3](#). Blank lines and whitespace.
- [Listing 4](#). An example of correct indentation.
- [Listing 5](#). Another sample script.

Introduction

You have learned how to write some simple programs and how to execute them interactively.

You have also learned how to capture simple programs in script files and how to execute those script files.

Now it is time to learn a little more about the mechanics of putting together a Python program.

Program construction

Conceptually, programs are constructed from statements, and statements are composed of expressions. In practice, you need to know how to construct statements from a physical viewpoint.

A Python program is divided into a number of *logical lines* . A logical line is constructed from one or more *physical lines*.

A physical line

A physical line ends with the character or characters used by your platform for terminating lines.

On Unix, this is the *linefeed* character. On Windows, it is the *carriage return* character followed by the *linefeed* character. On Macintosh, it is the *carriage return* character.

We refer to this in programming jargon as the *newline* character (*even though it is actually two characters on Windows systems*) .

Basically, a physical line is what you get when you type some characters into your text editor and press the *Enter* key, the *Return* key, or whatever you call that key on your keyboard. (*Actually you don't even need to type characters before pressing the Enter key, in which case you get a blank line.*)

This is the key that causes the cursor to go down to the next line and return to the left side of the editing window.

For example, when creating the text shown in [Figure 1](#) using a text editor on my Windows computer, I pressed the Enter key immediately following the --> for each line. This produced a *newline* in each case.

Figure 1 . Two physical lines.

```
I will end the line here -->
and end the next line here -->
```

A logical line

A *logical line* may be the same as a *physical line* , or line joining rules (*described later*) can be used to construct a logical line from two or more physical lines.

Statements and logical lines

A statement cannot cross logical line boundaries except where the syntax allows for the *newline* character, such as in *compound statements* . I will show you an example of a compound statement later in this module.

Comments

We learned about comments in an earlier module. To summarize, a comment starts with a hash character (#) that is not part of a string literal (*we also learned about string literals in an earlier module*) . The comment ends at the end of the physical line.

Explicit line joining

You can join two or more *physical lines* to produce a *logical line* , using the backslash character as shown in [Listing 1](#).

Listing 1 . Explicit line joining.

```
a = 3
b = 4
# construct c=a+b
c \
= \
a \
+ \
b
print(c)

# the output is 7
```

The backslash character

When a *physical line* ends in a backslash that is not part of a string literal or comment, that line is joined with the following *physical line* forming a single *logical line* .

The backslash and the following end-of-line character(s) are ignored by the interpreter and do not become part of the logical line.

In [Listing 1](#), the expression `c=a+b` is created by joining five consecutive *physical lines* to create one *logical line* .

For illustration only

Obviously, this is not how you would want to write a long script, but it is not unusual to break long expressions into two or more physical lines to make them fit onto a prescribed page width.

No spaces or comments allowed

You must make certain that no space characters follow the backslash.

You may not place a comment following the backslash, and a backslash does not continue a comment.

Otherwise illegal

A backslash is illegal elsewhere on a line except inside a string literal.

Implicit line joining

Expressions in parentheses, square brackets, or curly brackets can be split over more than one physical line without using backslashes as shown in [Listing 2](#).

Listing 2 . Implicit line joining.

Listing 2 . Implicit line joining.

```
a = 3
b = 4
c = (a # continue on next line
    + b)
print(c)

# the output is 7
```

You can also place a comment on a line that is being continued implicitly as shown in the third line of [Listing 2](#).

Also, you can indent the continuation line however you want. This is very useful for making the code more readable.

Blank lines

Lines containing only spaces, tabs, form feeds, and comments are ignored in scripts, but the behavior may be different in interactive mode, depending on the implementation.

An example of some spaces and blank lines is shown in [Listing 3](#).

Listing 3 . Blank lines and whitespace.

Listing 3 . Blank lines and whitespace.

```
a = 3
    # spaces and comment
b = 4
    # tab and comment
c = (a # continue on next line
    + b)
# a blank line follows

print(c)

# the output is 7
```

Indentation

In every other programming environment that I have worked with in the past, indentation is used strictly for cosmetic or readability purposes.

However, indentation is not used strictly for cosmetic or readability purposes in Python. It is such an important topic in Python that I have separated it into a major section in this module.

Indentation is used to determine grouping of statements

Unlike most other programming environments, physical indentation is used in Python to determine the grouping of statements.

A blessing and a curse

This can be both a blessing and a curse. The blessing is that it forces you to use proper indentation, which usually leads to more readable code.

The curse is that if you don't use proper indentation, your script probably won't behave properly.

No safety nets

There are few if any safety nets in Python (*such as the matching curly brackets used in C, C++, and Java*) to protect you from indentation errors.

To make matters worse, such errors often turn up as logical errors (*meaning that the script simply doesn't work correctly*) rather than interpreter errors (*meaning that the interpreter will tell you about the error*) .

An example of correct indentation

Although I haven't introduced you to the **if** statement yet, you probably have a fairly good idea what it is used for. I am going to use it to illustrate the proper indentation of a *group of statements* with the script in [Listing 4](#).

Note: Description of an "if" statement:

An **if** statement means that if some expression evaluates to **true** , the program will do something specific. Otherwise, the script won't do that specific thing.

Don't be too concerned if the logic of the script in [Listing 4](#) escapes you at this point in time. I will explain the use of the **if** statement in detail in a future module. The important thing here is to understand the grouping of statements.

Listing 4 . An example of correct indentation.

Listing 4 . An example of correct indentation.

```
A = 3
B = 4

if B > A:
    print(A) # begin group
    print(B)
    print(A + B) # end group
A = 6 # not part of above group
print(A)

#=====
#The output, which is not part of the script, is shown below.
3
4
7
6
```

Compare A with B and take appropriate action

The script in [Listing 4](#) compares the value of the variable **A** with the value of the variable **B** to determine if the value of **B** is greater than the value of **A** (*if B > A:*) .

If the value of **B** is greater than the value of **A** (*which it is in this case*) , the three indented print statements are executed. Otherwise, that group of three statements is bypassed.

A compound statement

The three statements shown with indentation are either all executed, or they are all bypassed. Hence, they behave as a group.

A group of statements like this is sometimes referred to as a *compound statement* (*a statement made up of two or more individual statements*) .

Group behavior

When the three indented statements are executed as a group, the values 3, 4, and 7 are printed on consecutive output lines by the three print statements in the group of statements, as shown in [Figure 2](#).

Figure 2 . Output from the script in Listing 4.

```
C:\jnk\6>echo off
3
4
7
6
Press any key to continue . . .
```

Then the value 6 is printed by the last statement in the script, which is not part of the group.
(Note that the last statement is not indented.)

Another sample script

Now, let's make a change to the script. The script shown in [Listing 5](#) is identical to the one in [Listing 4](#) except that I switched the values of **A** and **B** to cause the group of indented statements to be bypassed (*B is no longer greater than A*).

Listing 5 . Another sample script.

```
A = 4
B = 3

if B > A:
    print(A) # begin group
    print(B)
    print(A + B) # end group
A = 6 # not part of above group
print(A)

#=====
#The output, which is not part of the script, is shown below.
6
```

In this case, only one value (6) is printed because the three print statements in the group were bypassed as a group.

Members of the group

The important point here is that the three indented statements in [Listing 4](#) and [Listing 5](#) constitute a group because of their *common indentation level* .

An opinion

I personally don't like the idea of using indentation to create grouping. Although it sounds nice in theory, it can be very labor intensive in practice. Once you have written a script, one simple change can often require you to go back and modify the indentation level of almost every statement in the script.

I guess the good news is that this will encourage you to write your script as a series of short, concise independent modules rather than as a single long rambling script.

I am also concerned about the accessibility or lack thereof that grouping based on indentation level provides for blind and visually impaired students.

Indentation details

Leading whitespace

Leading whitespace (*spaces and tabs*) at the beginning of a *logical line* is used to compute the indentation level of the line. This, in turn, is used to determine the grouping of statements.

Tabs

My advice is to avoid the use of tabs altogether. Use spaces instead, and use the same number of spaces for each statement in the group.

However, if you must use tabs, you should go to the [Python Language Reference -- 2.1.8. Indentation](#) and make certain that you understand how Python deals with tabs.

The bottom line on indentation

Use spaces to cause the indentation level of all statements in a group of statements to be indented to the same level. Statements in a group are either all executed, or all bypassed by the

program logic.

I will provide a visualization of the behavior of indentation in the module titled [Itse1359-1065-Visualizing Python](#) in [this book](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1060-Syntax
- File: Itse1359-1060.htm
- Published: 10/14/14
- Revised: 03/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1060r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1060-Syntax.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1060-Syntax* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

Logical lines are constructed from one or more _____ (fill in the blank).

Go to [answer 1](#)

Question 2

What constitutes a physical line?

Go to [answer 2](#)

Question 3

Can statements cross logical line boundaries?

Go to [answer 3](#)

Question 4

What character is used for explicit line joining?

Go to [answer 4](#)

Question 5

Explain implicit line joining in your own words.

Go to [answer 5](#)

Question 6

What are the indentation rules for lines that have been implicitly joined?

Go to [answer 6](#)

Question 7

True or False? Indentation in Python is used for cosmetic purposes only.

Go to [answer 7](#)

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 7

False. Indentation is used in Python to determine the grouping of statements, which causes it to be critical to the success of a program.

Go back to [Question 7](#)

Answer 6

You can indent the continuation line however you want.

Go back to [Question 6](#)

Answer 5

Expressions in parentheses, square brackets, or curly brackets can be split over more than one physical line without using backslashes.

Go back to [Question 5](#)

Answer 4

The backslash character can be used to join two physical lines into one logical line.

Go back to [Question 4](#)

Answer 3

Yes, but only in those cases where the syntax allows for the *newline* character, such as in *compound statements* .

Go back to [Question 3](#)

Answer 2

A physical line is what you get when you type some characters into your text editor and press the *Enter* key, the *Return* key, or whatever it is called on your keyboard.

Go back to [Question 2](#)

Answer 1

Logical lines are constructed from one or more physical lines.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1060r-Review
- File: Itse1359-1060r.htm
- Published: 10/14/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1065-Visualizing Python

This module explains how to use a free educational code visualizer tool created by Philip Guo.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Recommendation](#)
- [Discussion and sample code](#)
 - [Open the online text editor](#)
 - [Three drop-down boxes](#)
 - [Beginning the visualization process](#)
 - [The red and green arrows](#)
 - [Output following three presses of the Forward button](#)
 - [Same situation, different code](#)
 - [Stepping through the code](#)
 - [The student torture program](#)
 - [Sharing the code visualizer with a tutor](#)
- [Run the programs](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on the use of the [Online Python Tutor](#), which includes a free educational *code visualizer tool* created by [Philip Guo](#).

This [code visualizer tool](#) helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this code visualizer tool, a teacher or student can write Python, Java, and JavaScript programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) The online text editor.
- [Figure 2.](#) Beginning the visualization process.
- [Figure 3.](#) Output following three presses of the Forward button.
- [Figure 4.](#) Same situation, different code.
- [Figure 5.](#) The student torture program.

Listings

- [Listing 1.](#) A sample program with proper indentation.
- [Listing 2.](#) Another sample program with proper indentation.

General background information

I touched on the use of the code visualizer in the earlier module titled [Itse1359-1050-Introduction to Scripts](#). I also touched on the code visualizer in the module titled [Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer](#) in [this book](#). The purpose of this module is to provide additional information related to the code visualizer.

When learning to program, it is often useful to be able to follow the execution of a program step-by-step. A free online tool called the [Online Python Tutor](https://pythontutor.com) makes this possible. The following quotation was copied from pythontutor.com.

*" [Online Python Tutor](https://pythontutor.com) is a free educational tool created by [Philip Guo](#) that helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this tool, a teacher or student can write **Python** , **Java** , and **JavaScript** programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs."*

Recommendation

I recommend that from this point forward in this course, you copy and paste every sample program into the code visualizer and observe the behavior of that code as you step through the program, one instruction at a time. I believe that doing that will help you to better understand Python programming.

The procedure for doing that is as follows:

- Open the [code visualizer](#).
- Copy the example program code into the online [text editor](#).
- Select Python 3.3 in the pull-down list.
- Click the button labeled **Visualize Execution** .
- Use the **Forward** and **Back** buttons to step forward and backward through your code, one instruction at a time, while observing the positions of the red and green arrows and observing the information that appears on the screen to the right of the code window.

Discussion and sample code

The earlier module titled [Itse1359-1060-Syntax](#) in [this book](#) presented and explained the two Python programs shown in [Listing 1](#) and [Listing 2](#). I will illustrate the use of the code visualizer with this pair of sample programs.

Listing 1. A sample program with proper indentation.

```
A = 3
B = 4

if B > A:
    print(A) # begin group
    print(B)
    print(A + B) # end group
A = 6 # not part of above group
print(A)
```

Listing 2. Another sample program with proper indentation.

Listing 2. Another sample program with proper indentation.

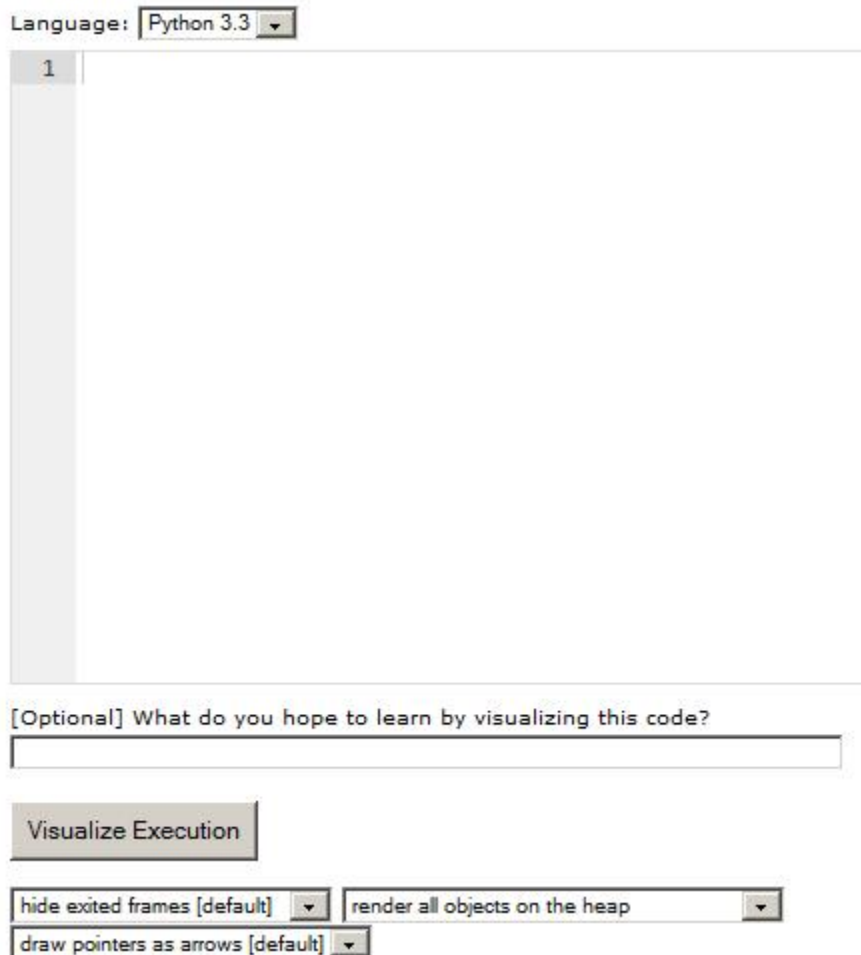
```
A = 4
B = 3

if B > A:
    print(A) # begin group
    print(B)
    print(A + B) # end group
A = 6 # not part of above group
print(A)
```

Open the online text editor

When you first open the online [text editor](#), a portion of the screen will look like [Figure 1](#).

Figure 1. The online text editor.



The blank portion at the top of [Figure 1](#) is the text editor portion of the screen.

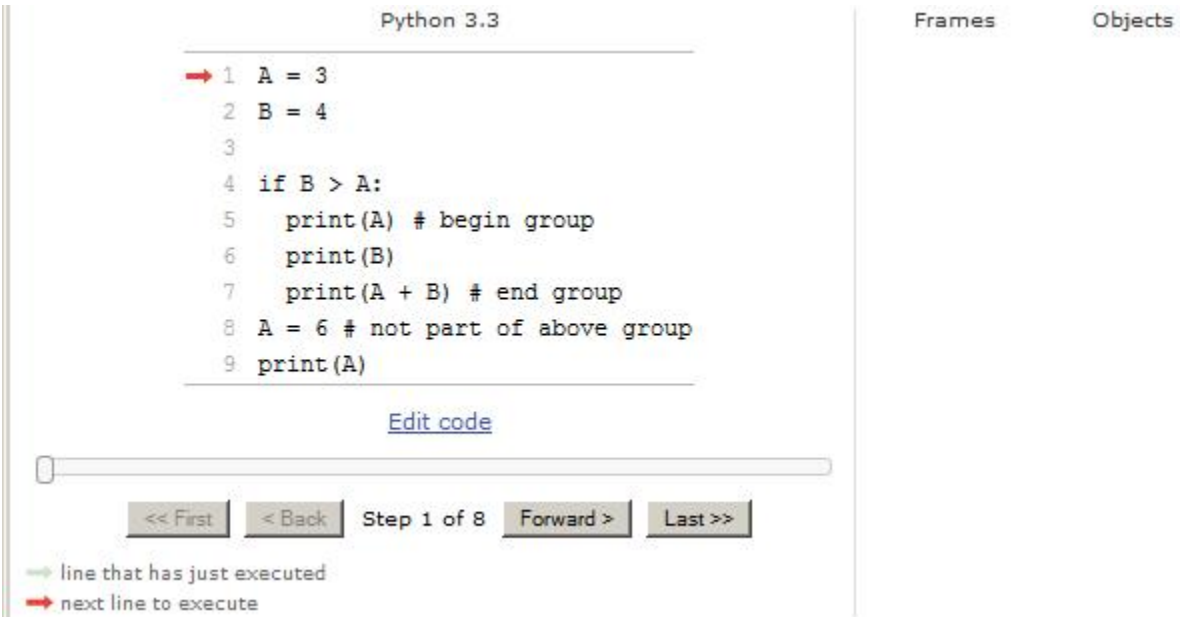
Three drop-down boxes

There are three drop-down boxes at the bottom of [Figure 1](#) that allow you to select different options. The different options in these boxes control the manner in which the output information is presented as you step through the program. In some cases, you might want to step through the same program more than once with different sets of options selected.

Beginning the visualization process

[Figure 2](#) shows the result of pasting the code from [Listing 1](#) into the text editor and pressing the button in [Figure 1](#) labeled **Visualize Execution**.

Figure 2. Beginning the visualization process.



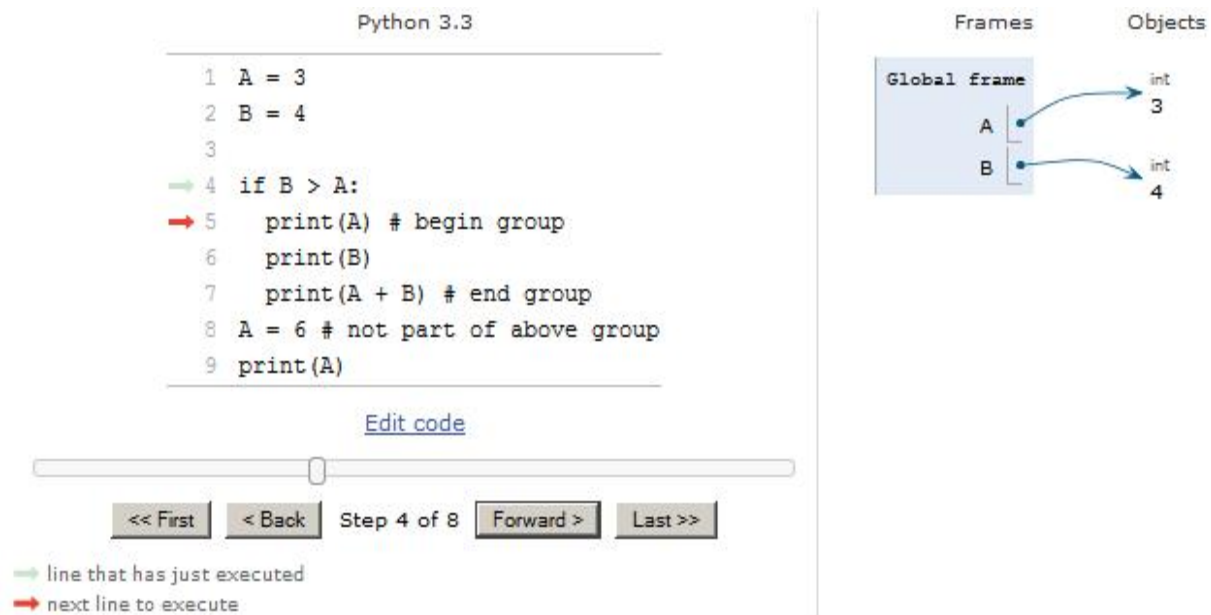
The red and green arrows

As indicated by the key in the bottom left corner of [Figure 2](#), two arrows will appear next to the code as you step through the code. A red arrow shows the instruction that will be executed the next time you press the **Forward** button. (*Pressing the **Back** button moves the red arrow back one step.*) The green arrow shows the instruction that was most recently executed. (*The green arrow isn't visible in [Figure 2](#) because it is underneath the red arrow at the beginning.*)

Output following three presses of the Forward button

[Figure 3](#) shows the result of pressing the **Forward** button three times.

Figure 3. Output following three presses of the Forward button.



We see from the green arrow on the left that the relational test in the **if** statement was the most recently executed instruction.

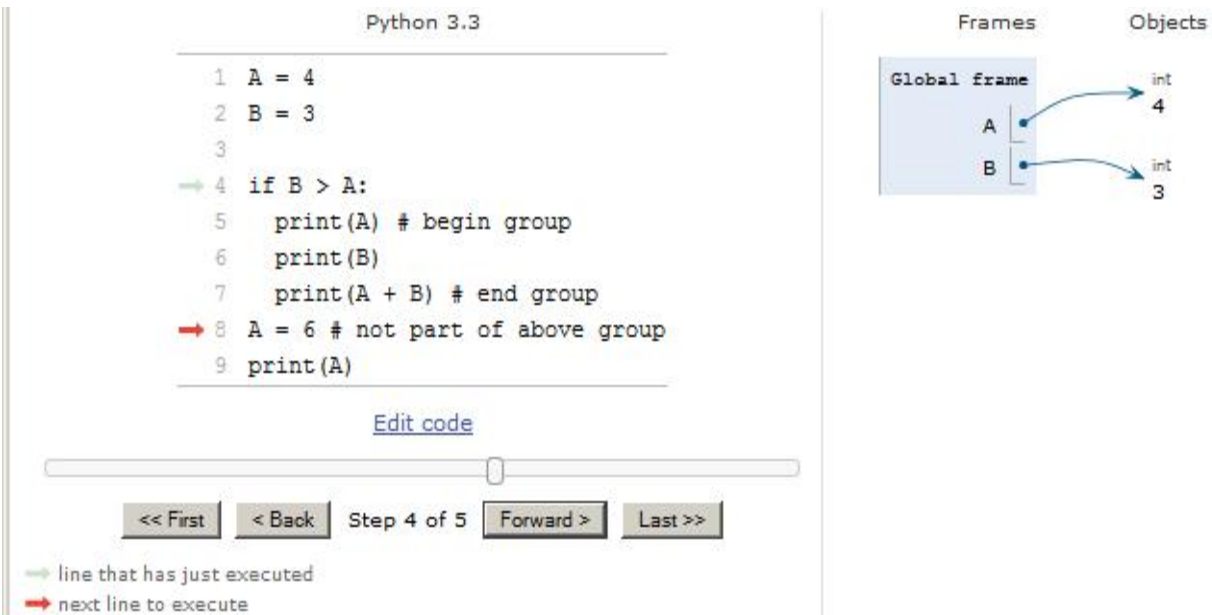
We see on the right that the variables named **A** and **B** contain references (*pointers*) to objects of type **int** containing values of **3** and **4** respectively.

Because **B** is greater than **A**, the relational test in the **if** statement (*pointed to by the green arrow*) returned true when that statement was executed. This means that the three statements in the indented block following the **if** statement will be executed. The red arrow points to the first statement in the indented block meaning that it will be executed the next time the user presses the **Forward** button.

Same situation, different code

[Figure 4](#) shows the same situation using the code from [Listing 2](#), instead of the code from [Listing 1](#).

Figure 4. Same situation, different code.



Note that the values stored in the variables named **A** and **B** were reversed relative to the code in [Figure 3](#). As a result, the **if** statement pointed to by the green arrow returned false instead of true when that statement was executed. Because of that, execution of the indented code block following the **if** statement will be skipped entirely and the red arrow now points to the statement immediately following that code block. That is the next statement that will be executed.

Stepping through the code

As you use the **Forward** and **Back** buttons to step through the code, one instruction at a time, you can observe the red and green arrows to gain a better understanding of the flow of control through the program. *(You will learn more about flow of control in future modules and you will probably find that the code visualizer is particularly useful when analyzing that code.)*

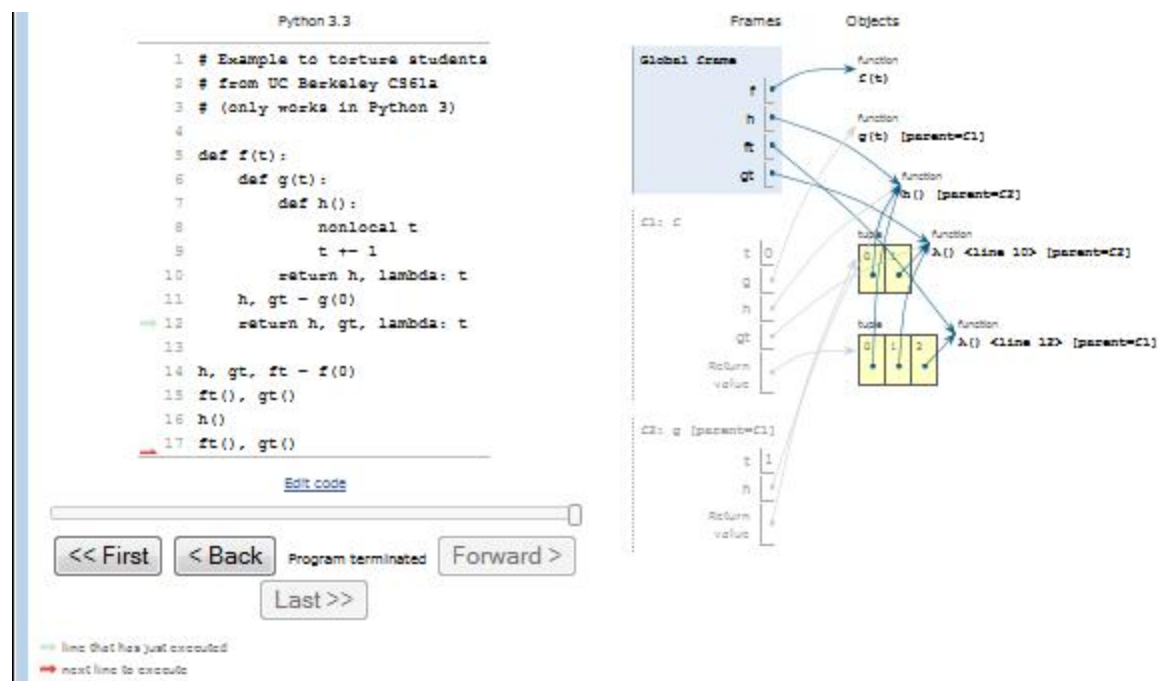
If you are in a hurry, you can press the **First** button and the **Last** button to cause the code visualizer to go the first or last instruction in one step.

The student torture program

There are links to quite a few sample programs below the drop-down boxes shown in [Figure 1](#). (I cropped them out of [Figure 1](#) to save space.) One of those sample programs is named **student torture**.

As a preview of things to come, [Figure 5](#) shows the result of opening that program in the code visualizer and pressing the **Last** button.

Figure 5. The student torture program.



Sharing the code visualizer with a tutor

Finally, there is another interesting sample program at <http://pythontutor.com/> that you can step through and observe its behavior. While you are there, be sure to note your ability to share the code visualizer with a friend or a tutor on another computer on the Internet and jointly analyze a program.

Run the programs

I encourage you to use the code in [Listing 1](#) and [Listing 2](#), along with the code visualizer to repeat the procedures explained in this module. Confirm that you get results similar to those shown in [Figure 3](#) and [Figure 4](#).

Change the options in the drop-down boxes and observe the results.

Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

As mentioned earlier, I strongly recommend that from this point forward in this course, you copy and paste every sample program into the code visualizer and observe the behavior of that code as you step through the program, one instruction at a time. I believe that doing that will help you to better understand Python programming.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1065-Visualizing Python
- File: Itse1359-1065.htm
- Published: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1070p-Preview-Strings Part 2

This module provides a preview of code that will be explained in more detail in the module titled Itse1359-1070-Strings Part 2.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1070-Strings Part 2](#) in [this book](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: The Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Program output.

Listings

- [Listing 1](#). Program code template.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output

You will be writing the code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output.

```
1.T
2.s
3.This
4.string
5.This
6.string
7.This is a string
8.This is a string
9.Empty:
10.tri
11.16
12.This is a string
```

Instructions

Copy the code template from [Listing 1.](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `###` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1.](#)

Listing 1. Program code template.

```
# File String01.py
# Rev 03/21/15
```

Listing 1. Program code template.

```
# Copyright 2015, R. G. Baldwin
# Illustrates indexing and
# slicing strings
#
#-----

aStr = "This is a string"
"""Print the output shown on Line 1."""
#??
"""Print the output shown on Line 2."""
#??
"""Print the output shown on Line 3."""
#??
"""Print the output shown on Line 4."""
#??
"""Print the output shown on Line 5 using
different syntax than 3 above."""
#??
"""Print the output shown on Line 6 using
different syntax than 4 above."""
#??
"""Print the output shown on Line 7 as the
concatenation of two slices
of aStr."""
#??
"""Print the output shown on Line 8 as a
single slice of aStr."""
#??
"""Print the output shown on Line 9 as the
concatenation of the word, the colon,
a space, and a slice of aStr."""
#??
"""Print the output shown on Line 10 as a
slice counting backwards from the
right end of aStr."""
```

Listing 1. Program code template.

```
#??  
"""Print the length of aStr as shown on Line  
11."""  
#??  
  
print("12." + aStr)
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1070-Strings Part 2](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1070p-Preview-Strings Part 2
- File: Itse1359-1070p.htm
- Published: 03/21/15
- Revised: 03/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1070-Strings Part 2

This module will expand your knowledge of Python strings, and in addition will introduce you to some concepts that will be useful with other data types as well: indexing and slicing.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Indexing](#)
 - [A practical example](#)
 - [Assigning an ordinal index](#)
 - [Using an ordinal index](#)
 - [Index values automatically assigned](#)
 - [Slicing](#)
- [A sample program](#)
 - [The program listing](#)
 - [Will discuss in fragments](#)
- [Interesting code fragments](#)
 - [Extract a single character](#)
 - [Create a string object to work with](#)
 - [Visualizing the string object](#)

- [Index values always begin with zero](#)
- [Display the character at index 3](#)
- [Is this the fourth character?](#)
- [An important and potentially confusing point](#)
- [Not like eggs](#)
- [A simple slice](#)
 - [Slice notation](#)
 - [The end is non-inclusive](#)
 - [Extract the first word in the string](#)
 - [Extract the last word in the string](#)
- [A more complicated slice](#)
 - [Omitting the first index](#)
 - [Omitting the second index](#)
 - [Print the entire string](#)
 - [Print an empty string](#)
 - [Negative indices](#)
 - [Eliminating confusion](#)
- [Getting the length of a string](#)
- [The complete output](#)
- [A string is Immutable](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It will expand your knowledge of strings, and in addition will introduce you to some concepts that will be useful with other data types as well: *indexing* and *slicing* .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. However, some of the Figures contain images that may not be accessible.)

Figures

- [Figure 1](#). Visualizing the string object.

Listings

- [Listing 1](#). Extract a single character.
- [Listing 2](#). A simple slice.
- [Listing 3](#). Omitting the first index.
- [Listing 4](#). Omitting the second index.
- [Listing 5](#). Print the entire string.
- [Listing 6](#). Print an empty string.
- [Listing 7](#). Negative indices.
- [Listing 8](#). Getting the length of a string.
- [Listing 9](#). The script named String01.py.

Introduction

What you have learned

You have learned how to write some simple Python programs and to execute them interactively.

You have learned how to capture simple programs in script files and to execute those script files.

You have learned how to construct programs, including the indentation required by Python.

You have also learned some of the fundamental concepts involving strings.

What you will learn

This module will expand your knowledge of strings and in addition will introduce you to some concepts that will be useful with other data types as well: *indexing* and *slicing* .

Indexing

According to [WolframMathWorld](#) , "... an [ordinal number](#) is an adjective which describes the numerical position of an object, e.g., first, second, third, etc."

A practical example

Many years ago when I did a tour of duty as an enlisted man in the U.S. Air Force, the drill sergeant had a habit of lining us up and telling us to "*count off*."

What that meant was that the first person in the line called out the number **one** , the person behind him called out the number **two** , the person behind him called out the number **three** , etc.

(Since learning about computer programming, I now wonder if the first person should have called out zero. I notice that the first

case of Ebola in the United States is being identified by the CDC as case 0.)

Assigning an ordinal index

I'm sure the drill sergeant didn't realize that what he was doing was assigning an ordinal index value to each person in the line (*and neither did I realize it at the time*) .

Using an ordinal index

Even though he didn't know the technical details of ordinal indices, he didn't have any difficulty yelling, "Number six, wash dishes, number fourteen, peel potatoes, number twenty-two, carry out the garbage, etc."

That is what using an index is all about -- using an ordinal index to select an item. As you will see in a future module titled *Itse1359-1080-Lists Part 1* , this is also referred to as a *subscription* in Python. In the context of this module, indexing is the process of assigning an ordinal index value to each data item contained in some sort of a container.

In other words, we assign an ordinal number to each item, which describes the numerical position of the item in the container.

Note:

A dozen eggs:

For example, if you were very careful, you could use a felt tip pen to assign an ordinal index to each of the twelve eggs contained in a carton containing a dozen eggs. (*Should you start with zero or one?*)

Then you could extract the egg whose index value is 9 from the container and eat it for breakfast.

Having assigned the index, we can use that index to access the data item corresponding to that index, as in *"Number six, wash dishes."*

Index values automatically assigned

In this module, we will be using the index values that are automatically assigned to the characters in a string for the purpose of accessing those characters, both individually, and in groups.

Slicing

Here is what [Magnus Lie Hetland](#) has to say on the topic of slicing (*and indexing as well.*) Although this quotation was taken from a discussion of lists, it applies equally well to strings. (*Note that some material was deleted from the quotation for brevity.*)

Note:

According to Magnus Lie Hetland:

"One of the nice things about lists is that you can access their elements separately or in groups, through indexing and slicing.

Indexing is done (*as in many other languages*) by appending the index in brackets to the list. (*Note that the first element has index 0*) ...

(*This is the answer to the question about the first egg -- Baldwin*)

Slicing is almost like indexing, except that you indicate both the start and stop index of the result, with a colon (":") separating them: ...

Notice that the end is non-inclusive. If one of the indices is dropped, it is assumed that you want everything in that direction. i.e. `list[:3]` means "every element from the beginning of list up to element 3, non-inclusive."

...

`list[3:]` would, on the other hand, mean "every element from list, starting at element 3 (*inclusive*) up to, and including, the last one."

For really interesting results, you can use negative numbers too: `list[-3]` is the third element from the end of the list..."

A sample program

I will illustrate indexing and slicing of strings using a sample program contained in a script file named **String01.py** .

The program listing

A complete listing of the program, *(along with the output produced by the program)* , is provided in [Listing 9](#) near the end of the module.

Will discuss in fragments

I will discuss the program in fragments, illustrating particular aspects of indexing and slicing in each fragment. This is a scheme that I will use frequently in this set of tutorial modules.

Interesting code fragments

Extract a single character

A single character can be extracted from a string by referring to the string and enclosing the index of the character in square brackets, as shown in the code fragment in [Listing 1](#).

Listing 1 . Extract a single character.

Listing 1 . Extract a single character.

```
aStr = "This is a string"
print(aStr[0]) #print T
print(aStr[3]) #print s
```

(Note that this is a fragment from a script file, not from an interactive Python program.)

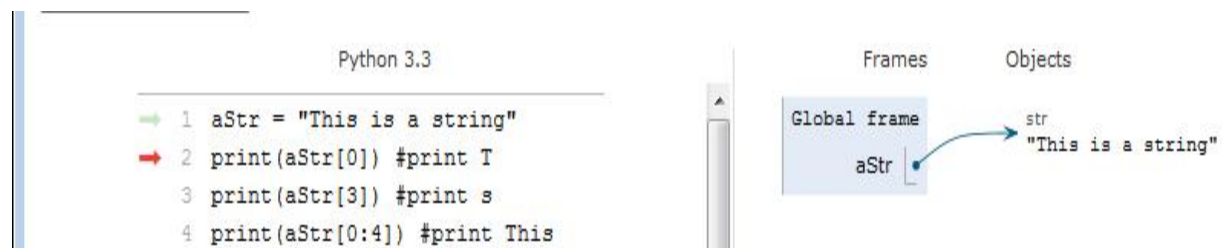
Create a string object to work with

The fragment in [Listing 1](#) creates a string object and assigns it to a variable named **aStr** . From this point forward, the contents of the string can be accessed by referring to the variable.

Visualizing the string object

[Figure 1](#) shows a visualization of the string object. This visualization was produced using the [Online Python Tutor](#) .

Figure 1. Visualizing the string object.



The green arrow in [Figure 1](#) indicates that the first statement in the program is the most recently executed statement in this visualization. (*This is the statement that created the string object.*) The diagram on the right shows

that the variable named **aStr** contains a reference to an object of type **str** , which contains the value: *"This is a string"* .

This is a very simple diagram but it shows a pattern that you need to remember. Variables in Python contain references to objects. Those objects may be *"dumb"* objects intended solely to encapsulate a value (such as type **int**) or they may be *"smart"* objects (such as type **str**) that can not only encapsulate data values but can also encapsulate methods (such as **capitalize()**) to manipulate those data values in various ways.

*(Note that even the dumb objects are not completely dumb. For example, all of the numeric types such as **int** know how to participate in arithmetic operations, how to negate themselves, and how to do a few other things that might be expected of a numeric.)*

Index values always begin with zero

Unlike eggs and Air Force enlisted men, the first character in a string is always located at index 0, as in **aStr[0]** .

Thus, the second statement in the fragment extracts and prints the **T** , which is the first character in the word **This** .

Display the character at index 3

Similarly, the last statement in the fragment in [Listing 1](#) extracts and prints the **s** from index position 3, as in **aStr[3]** . The character at this index position is the **s** that ends the word **This** .

The last statement is equivalent to the following request, *"Will the character at index position 3 please display yourself on the screen."*

Is this the fourth character ?

You would probably refer to this as the fourth character, and you would be correct if you did. The character at index 3 is the fourth character in the string. The character at index 0 is the first character in the string. First does not equate to index 1. You need to think about this, because this can be a very confusing topic for new programmers.

An important and potentially confusing point

At the risk of becoming boring, there is an important point here that you might as well get used to right now.

The `s` is located at index value 3. However, according to the way you are probably accustomed to counting, this is actually the fourth character in the string. You might be inclined to refer to this character as character number 4.

This is because index values always begin with zero, while you are probably accustomed to counting things beginning with one, not zero.

Not like eggs

Regardless of whether you begin with zero or one, if you access the egg at index value 4 in the container and eat it for breakfast, it cannot be accessed again (*because it will be gone*) .

However, if you access the character at index value 4 in the string and use it for some purpose, what you really use is a copy of the character. It is still there and it can be accessed again.

(Some data containers do allow for the removal of data elements in much the same sense that we can remove an egg from its container. However, a string is not such a container.)

A simple slice

The fragment in [Listing 2](#) cuts a couple of slices out of the string that was created in [Listing 1](#) and displays them on the screen.

Listing 2 . A simple slice.

```
print(aStr[0:4]) #print This  
print(aStr[10:16]) #print string
```

Slice notation

The slice notation uses two index values separated by a colon, as shown in [Listing 2](#).

The end is non-inclusive

As was indicated in the earlier quotation, "*... the end is non-inclusive.*" This means that the character whose index value is the number following the colon is not included in the slice.

Extract the first word in the string

Thus, the first statement in [Listing 2](#) containing the reference **aStr[0:4]** extracts and prints the character sequence beginning with index value 0 and ending with index value 3 (*not 4*) . This causes the word **This** to be extracted and printed.

Extract the last word in the string

Similarly, the second statement in the above fragment containing the reference `aStr[10:16]` extracts and prints the characters having index values from 10 through 15, inclusive (*not 16*) . This causes the word **string** to be extracted and printed.

A more complicated slice

Omitting the first index

If you omit the first index value, as shown in [Listing 3](#), it defaults to the value zero.

Listing 3 . Omitting the first index.

```
print(aStr[:4]) #print This
```

Therefore, the statement in [Listing 3](#) extracts and prints the first word in the string, which is **This** .

Omitting the second index

If you omit the second index, as shown in [Listing 4](#), it defaults to a value that includes the last character in the string.

Listing 4 . Omitting the second index.

```
print(aStr[10:]) #print string
```

Thus, the statement in [Listing 4](#) extracts and prints the last word in the string, which is **string** .

Print the entire string

[Listing 5](#) shows two different ways to extract and print the entire string. I won't comment on this, but will leave the analysis as an exercise for the student.

Listing 5 . Print the entire string.

```
#print the entire string  
print(aStr[:5] + aStr[5:])  
print(aStr[:100])
```

(Hint: Remember that the plus sign when used with strings is the string concatenation operator.)

Print an empty string

There are several ways that you can specify the index values that will produce an empty string. One of those ways is shown following the plus sign in [Listing 6](#).

Listing 6 . Print an empty string.

```
#print an empty string  
print("Empty: " + aStr[16:100])
```

In [Listing 6](#), both index values are outside the bounds of the index values of the characters in the string, which range from 0 through 15 inclusive.

Negative indices

Although it can be a little confusing, negative index values can be used to count from the right, as shown in [Listing 7](#).

Listing 7 . Negative indices.

Listing 7 . Negative indices.

```
#count from the right  
print(aStr[-5:-2]) #print tri
```

This fragment extracts and prints the characters **tri** from the word **string** , which is the last word in the string.

Eliminating confusion

Once you allow negative indices for slicing, things can become very confusing. The following explanation of how indices work with slicing is attributed to [Guido van Rossum](#) , the author of the Python programming language.

In this example, Mr. van Rossum is referring to a five-character string with a value of " **HelpA** ".

Note:

Eliminating confusion

The best way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n, for example:

+	-	-	-	+	-	-	-	+	-	-	-	+
	H		e		l		p		A			
+	-	-	-	+	-	-	-	+	-	-	-	+
0		1		2		3		4		5		
-5		-4		-3		-2		-1				

The first row of numbers gives the position of the indices 0...5 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively. For nonnegative indices, the length of a slice is the difference of the indices, if both are within bounds, e.g., the length of word[1:3] is 2.

Hopefully, this explanation will help you to understand and to remember how index values are used for the extraction of substrings from strings using slicing.

Getting the length of a string

And finally, a built-in function named **len()** can be used to determine the number of characters actually contained in a string as shown in [Listing 8](#).

Listing 8 . Getting the length of a string.

```
#get the length of the string
print(len(aStr))
```

For the example string used in this module, [Listing 8](#) gets and prints the length of the string as 16.

If you count the characters in the string (*beginning with 1*) , you will conclude that there are 16 characters in the string.

Note the difference between the number of characters and the maximum index value

For a string containing 16 characters, the valid index values range from 0 through 15 inclusive.

The complete output

This Python script file produces the output shown in the lower portion of [Listing 9](#).

A string is immutable

There is one more point that needs to be made here. Although you can use indexing and slicing to access the characters in a string, you cannot use indexing and slicing to assign new character values to those characters.

This is because a Python string is *immutable* . In other words, after it is created, it cannot be modified.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1070-Strings Part 2
- File: Itse1359-1070.htm
- Published: 10/15/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program follows is shown in [Listing 9](#).

Listing 9 . The script named String01.py.

```
# File String01.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates indexing and
# slicing strings
```

Listing 9 . The script named String01.py.

```
#
#-----

aStr = "This is a string"
print(aStr[0]) #print T
print(aStr[3]) #print s
print(aStr[0:4]) #print This
print(aStr[10:16]) #print string
print(aStr[:4]) #print This
print(aStr[10:]) #print string

#print the entire string
print(aStr[:5] + aStr[5:])
print(aStr[:100])

#print an empty string
print("Empty: " + aStr[16:100])

#count from the right
print(aStr[-5:-2]) #print tri

#get the length of the string
print(len(aStr))

#=====
#This script produces the following output,
#which is not part of the script.
T
s
This
string
This
string
This is a string
This is a string
```

Listing 9 . The script named String01.py.

```
Empty:  
tri  
16
```

-end-

Itse1359-1070r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1070-Strings Part 2.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1070-Strings Part 2* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1 .

What is an ordinal number?

Go to [answer 1](#)

Question 2

What is indexing?

Go to [answer 2](#)

Question 3

True or False? We assign index values to the characters in a string.

Go to [answer 3](#)

Question 4

What is the syntax for accessing a character at a particular index in a string?

Go to [answer 4](#)

Question 5

What is the syntax for accessing a substring from a string using slicing?

Go to [answer 5](#)

Question 6

True or False? The second index of a slice is inclusive.

Go to [answer 6](#)

Question 7

What is the default value for the first index if you omit the first index value in a slice?

Go to [answer 7](#)

Question 8

What is the default value for the second index if you omit the second index value in a slice?

Go to [answer 8](#)

Question 9

What is the purpose of negative slice indices?

Go to [answer 9](#)

Question 10

What is the name and syntax of the function that can be used to find the length of a string?

Go to [answer 10](#)

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 10

The function is named **len** . It requires one parameter, which is the name of the string as in:

len(theString)

Go back to [Question 10](#)

Answer 9

Negative indices can be used to count from the right end of the string toward the beginning of the string on the left.

Go back to [Question 9](#)

Answer 8

The default value is the length of the string (*the actual number of characters in the string*) , which is greater than the index of the last character in the string.

Go back to [Question 8](#)

Answer 7

The default value is zero (0).

Go back to [Question 7](#)

Answer 6

False. The character whose index value matches the second index of a slice is not included in the slice.

Go back to [Question 6](#)

Answer 5

Include both the start and stop index in square brackets, separated by a colon, as in `aStr[10:16]` .

Go back to [Question 5](#)

Answer 4

Refer to the string, and include the index value in square brackets, as in `aStr[3]` .

Go back to [Question 4](#)

Answer 3

False. Index values are automatically assigned to the characters in a string.

Go back to [Question 3](#)

Answer 2

Indexing is the process of assigning an ordinal index value to each data item contained in some sort of a container.

Go back to [Question 2](#)

Answer 1

An ordinal number is an adjective that describes the numerical position of an object, e.g., first, second, third, etc.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1070r-Review
- File: Itse1359-1070r.htm
- Published: 10/15/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales

nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1080pa-Preview-01-Lists Part 1

This module provides a preview of code that will be explained in more detail in the module titled Itse1359-1080p-Lists Part 1.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of some of the code that will be explained in more detail in the module titled [Itse1359-1080-Lists Part 1](#) in [this book](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: The Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Program output for example list.

Listings

- [Listing 1](#). Program code template for example list.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for example list.

```
1. A string
2. [3.14, 59, 'A string']
3. [3.14, 59, 'A string', 1024]
4. [3.14, 59, 'A string', 1024]
```

Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `###` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for example list.

Listing 1. Program code template for example list.

```
theList = [3.14,59,"A string",1024]
"""Print the output shown on Line 1."""
#??
"""Print the output shown on Line 2."""
#??
"""Print the output shown on Line 3 using a
slice of theList."""
#??

print("4.",theList)
```

Answers in the back of the book

If you get stuck and are unable to successfully complete the modifications, you will find the correct code in the module titled [Itse1359-1080-Lists Part 1](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1080pa-Preview-01-Lists Part 1
- File: Itse1359-1080pa.htm
- Published: 03/21/15
- Revised: 03/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1080pb-Preview-02-Lists Part 1

This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1080p-Lists Part 1](#).

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output](#)
 - [Instructions](#)
- [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of some of the code that will be explained in more detail in the module titled [Itse1359-1080-Lists Part 1](#) in [this book](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: The Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Program output for concatenated lists.

Listings

- [Listing 1](#). Program code template for concatenated lists.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for concatenated lists.

```
1. [3.14, 59, 'A string', 1024]
2. [2, 4, 6, 16]
3. [3.14, 59, 'A string', 1024, 2, 4, 6, 16]
4. end
```

Instructions

Copy the code template from [Listing 1](#). into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `###` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for concatenated lists.

Listing 1. Program code template for concatenated lists.

```
listA = [3.14,59,"A string",1024]
listB = [2,4,6,16]

"""Print the output shown on Line 1."""
#??
"""Print the output shown on Line 2."""
#??
"""Create and print the output shown on Line
3."""
#??

print("4.", "end")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1080-Lists Part 1](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1080pb-Preview-02-Lists Part 1
- File: Itse1359-1080pb.htm
- Published: 03/25/15

- Revised: 03/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1080pc-Preview-03-Lists Part 1

This module provides a preview of code that will be explained in more detail in the module titled Itse1359-1080p-Lists Part 1.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1080-Lists Part 1](#) in [this book](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: The Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Program output for mutable lists.

Listings

- [Listing 1](#). Program code template for mutable lists.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output

You will be writing the code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1 . Program output for mutable lists.

```
1. [3.14, 59, 'A string', 1024]
2. [3.14, 59, 'New string', 1024]
3. [3.14, 59, 'New string', 2048]
4. [3.14, 59, 0.99999, 2048]
5. end
```

Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `###` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for mutable lists.

Listing 1. Program code template for mutable lists.

```
listA = [3.14,59,"A string",1024]

"""Print the output shown on Line 1."""
#??

"""Create and print the output shown on Line
2."""
#??

"""Create and print the output shown on Line
3."""
#??

"""Create and print the output shown on Line
4."""
#??

print("5.", "end")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1080-Lists Part 1](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#) above.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1080p-Preview-Lists Part 1
- File: Itse1359-1080p.htm
- Published: 03/25/15
- Revised: 03/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1080-Lists Part 1

This module extends our Python knowledge to a new type of data called a list. This type of data, along with a string, is referred to as a sequence.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
 - [What you have learned](#)
 - [Introducing lists](#)
- [A subscription](#)
 - [Description](#)
 - [The primary](#)
 - [A clarification](#)
 - [The expression list](#)
 - [Negative integers](#)
 - [Example of a negative index](#)
 - [More negative-integer rules](#)
 - [Good and bad negative integers](#)
 - [The character type](#)
- [A sequence](#)
- [A mapping](#)
 - [A real dictionary example](#)
 - [Is this a definition or a value?](#)

- [No mention of the Python language](#)
- [What about the expression list?](#)
- [Slicing](#)
- [A mutable sequence](#)
- [A list](#)
- [Some sample programs](#)
- [Creating, indexing, and slicing lists](#)
 - [An example list](#)
 - [Print an item using a subscription](#)
 - [Print some slices](#)
 - [Program output](#)
 - [Visualize sliced lists](#)
 - [Create new lists by slicing an existing list](#)
 - [Change an item in the original list](#)
 - [Lists can be concatenated](#)
 - [A concatenation program](#)
 - [Concatenation program output](#)
 - [Visualize concatenated lists](#)
 - [Create a new list by concatenating two existing lists](#)
 - [Change an element in an original list](#)
 - [Lists are mutable](#)
 - [Replace a string with another string](#)
 - [Multiply an integer element by two](#)
 - [Replace a string by a float](#)
 - [List modification program output](#)

- [More to come](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It extends our Python knowledge to a new type of data called a list. This type of data, along with a **string**, is referred to as a *sequence*.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: many of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Program output.
- [Figure 2](#). Visualize sliced lists.
- [Figure 3](#). Change an item in the original list.
- [Figure 4](#). Concatenation program output.
- [Figure 5](#). Visualize concatenated lists.
- [Figure 6](#). Change an element in an original list.
- [Figure 7](#). List modification program output.

Listings

- [Listing 1](#). A string as a primary.
- [Listing 2](#). Example of a negative index.
- [Listing 3](#). Good and bad negative integers.
- [Listing 4](#). An example list.
- [Listing 5](#). Lists can be concatenated.
- [Listing 6](#). Lists are mutable.

Introduction

What you have learned

In an earlier module you learned how to *index* and *slice* strings.

Introducing lists

In this module, we will extend that knowledge to a new type of data called a *list*. This type of data, along with a **string**, is referred to as a *sequence*.

First, however, I will nail down some terminology from the [Python Language Reference](#), [The Python Tutorial -- 3.1.3. Lists](#), and [The Python Standard Library -- 4.6. Sequence Types -- list, tuple, range](#).

A subscription

I referred to a *subscription* as an *index* in the earlier module titled [Itse1359-1070-Strings Part 2](#) in [this book](#).

The following discussion of a *subscription* is based on the [Python Language Reference -- 6.3.2. Subscriptions](#).

Description

A *subscription* selects an item of a *sequence* (*string* , *tuple* or *list*) or *mapping* (*dictionary*) object, **as in the following** :

primary "[" **expression_list** "]"

I discussed strings earlier. I will discuss lists in this and the next module. I will discuss tuples *and* mappings later.

The primary

According to the [Python Language Reference -- 6.3.2. Subscriptions](#)

"The primary must evaluate to an object that supports subscription, e.g. a list or dictionary. User-defined objects can support subscription by defining a `__getitem__()` method."

For example, in the earlier module titled [Itse1359-1070-Strings Part 2](#), the *primary* was a reference to an object of type **str** where the reference was stored in a variable named **aStr** , as shown in [Listing 1](#). A diagram showing the variable and the object was shown in **Figure 1** in that module.

Listing 1 . A string as a primary.

```
aStr = "This is a string"
print(aStr[0]) #print T
print(aStr[3]) #print s
```

A clarification

This terminology can be a little confusing. Perhaps the following will help to clarify things.

An [ordinal number](#) is an adjective that describes the numerical position of an object, e.g., first, second, third, etc.

Indexing is the process of assigning an ordinal number to each data item contained in some sort of a container.

The **index** is the value of an ordinal number that has been assigned to an item.

A **subscription** is the process of selecting an item of a *sequence* (*string* , *tuple* or *list*) or *mapping* (*dictionary*) object.

The expression_list

The earlier [description](#) included the term **expression_list** .

The [Python Language Reference -- 6.3.2. Subscriptions](#) states:

"If the primary is a sequence, the expression_list must evaluate to an integer or a slice..."

An example of an integer expression_list

In the string example in [Listing 1](#), the integer was 0 in one case and 3 in the other case.

Negative integers

If the (*index*) value is negative, the *length* of the sequence is added to it to obtain the actual index used to access the sequence.

Example of a negative index

For example, **aSequence[-1]** selects the last item of the sequence.

This is illustrated in the script shown in [Listing 2](#), which prints the last character in a string.

Listing 2 . Example of a negative index.

```
aStr = "xyz"  
print(aStr[-1])  
  
# the output is z
```

The length of the string is 3.

The sum of the length and the negative index is 2.

The character in the string at an index value of 2 is "z".

More negative-integer rules

The value resulting from adding the length of the sequence to the specified index value must be a non-negative integer less than the number of items in

the sequence.

Then, the subscription selects the item whose index is that value (*counting from zero*) .

Good and bad negative integers

This is illustrated by the script in [Listing 3](#), which shows both valid and invalid negative index values. (*The index value of -4 violates the above rule, thus producing an `IndexError`.*)

Listing 3 . Good and bad negative integers.

Listing 3 . Good and bad negative integers.

```
aString = "xyz"
print(aString[-1])
print(aString[-2])
print(aString[-3])
print(aString[-4])

#The output from this script is shown below.
z
y
x
Traceback (most recent call last):
  File "1359-1080-03.py", line 9, in <module>
    print(aString[-4])
IndexError: string index out of range
```

The character type

Unlike many other programming languages, there is no *character* type in Python. Rather, a string's items are characters. A character is not a separate data type but is a string of exactly one character.

A sequence

The [Python Language Reference -- 6.3.2. Subscriptions](#) indicates that:

Python provides three kinds of sequences: *strings* , *lists* , and *tuples* .

[The Python Standard Library -- 4.6. Sequence Types -- list, tuple, range](#) states:

"There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections."

Further down the page, the [Python Standard Library -- 4.7. Text Sequence Type -- str](#) states:

"Textual data in Python is handled with **str** objects, or strings. Strings are immutable sequences of Unicode code points."

From that, we can conclude that at least the following are *sequence types* in Python:

- lists
- tuples
- range objects
- strings

I discussed strings in previous modules. I will discuss lists in this module, and I will discuss tuples and range objects in future modules.

A mapping

The [Python Standard Library - 4. Built-in Types](#) states:

"The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions."

A *mapping* is a structure in which values are stored and retrieved according to a *key* . This is often called a dictionary, because it behaves similarly to a common dictionary.

The [Python Standard Library -- 4.10. Mapping Types -- dict](#) states:

"A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary."

A real dictionary example

For example, if I want to know about the word Python, I get out my Webster's Seventh New Collegiate Dictionary, (*which is at least 55 years old*) , and I look up the word *python* . In this case, *python* is the key.

The (*approximate*) value associated with the key is, "*... monstrous serpent killed by Apollo ...*"

Is this a definition or a value ?

In normal conversation, we frequently refer to the value obtained from a common printed dictionary as the *definition* of the word. However, definition usually means something completely different in computer jargon.

In computer jargon, we refer to it as the *value* associated with the word (*key*). Thus, a *mapping* , is a structure that associates *values* with *keys* .

So, a common printed dictionary is an example of a mapping.

No mention of the Python language

Since my dictionary is much older than the Python programming language, *(and probably older than the author of Python)* I wouldn't expect to find anything about Python programming there.

What about the `expression_list` ?

According to the [Python Language Reference -- 6.3.2. Subscriptions](#),

"If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key."

In other words, the `expression_list` must evaluate to an object whose value is one of the keys of the mapping *(such as the word `python` in my common dictionary example)* .

Then the subscription selects the value in the mapping that corresponds to that key.

In other words, the system looks up the word in the dictionary and returns the value that corresponds to that word.

I will have more to say about *mappings* in a future module.

Slicing

I discussed *slicing* at some length in the earlier module titled [Itse1359-1070-Strings Part 2](#). I will simply refer you back to that module for the discussion on slicing.

A mutable sequence

According to the [Python Language Reference -- 3.2. The standard type hierarchy](#):

Note: Mutable sequences:

Mutable sequences can be changed after they are created.

There are currently two intrinsic mutable sequence types:

- Lists - The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)
- Byte Arrays - A bytearray object is a mutable array. They are created by the built-in bytearray() constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable bytes objects.

This module does not discuss **ByteArrays** .

A list

According to [The Python Tutorial -- 3.1.3. Lists](#):

"Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between

square brackets. Lists might contain items of different types, but usually the items all have the same type.

Like strings (and all other built-in sequence type), lists can be indexed and sliced.

All slice operations return a new list containing the requested elements."

Some sample programs

Creating, indexing, and slicing lists

A list can be written as a sequence of comma-separated values (*items*) surrounded by square brackets.

Lists can also be nested within other lists.

List items do not all have to be of the same type.

An example list

The Python script shown in [Listing 4](#) creates a simple list containing four elements of different types. The types of the elements are respectively, a *float* value, an *integer* , a *string* , and another *integer* .

Listing 4 . An example list.

```
# Illustrates creating,  
# indexing, and slicing lists.  
#  
#-----  
  
theList = [3.14,59,"A string",1024]  
print("Print index value 2")  
print(theList[2])  
print("Print a short slice")  
print(theList[0:3])  
print("Print the entire list")  
print(theList[:100])
```

Print an item using a subscription

After creating the list, the program uses a subscription (*index*) to extract and print the value at index 2 (*remember the first item is at index 0*) .

Print some slices

Then it uses the slice notation to extract and print two different slices from the list.

The first slice extracts and prints the elements from index 0 through index 2 inclusive. (*Remember, the items selected by a slice do not include the index specified by the upper limit value, which is 3 in this case.*)

The second slice extracts and prints the entire list. If you don't understand these two slices, go back and review the module titled [Itse1359-1070-](#)

[Strings Part 2](#) where I discuss slicing in detail.

Program output

The output from this program is shown in [Figure 1](#).

Figure 1 . Program output.

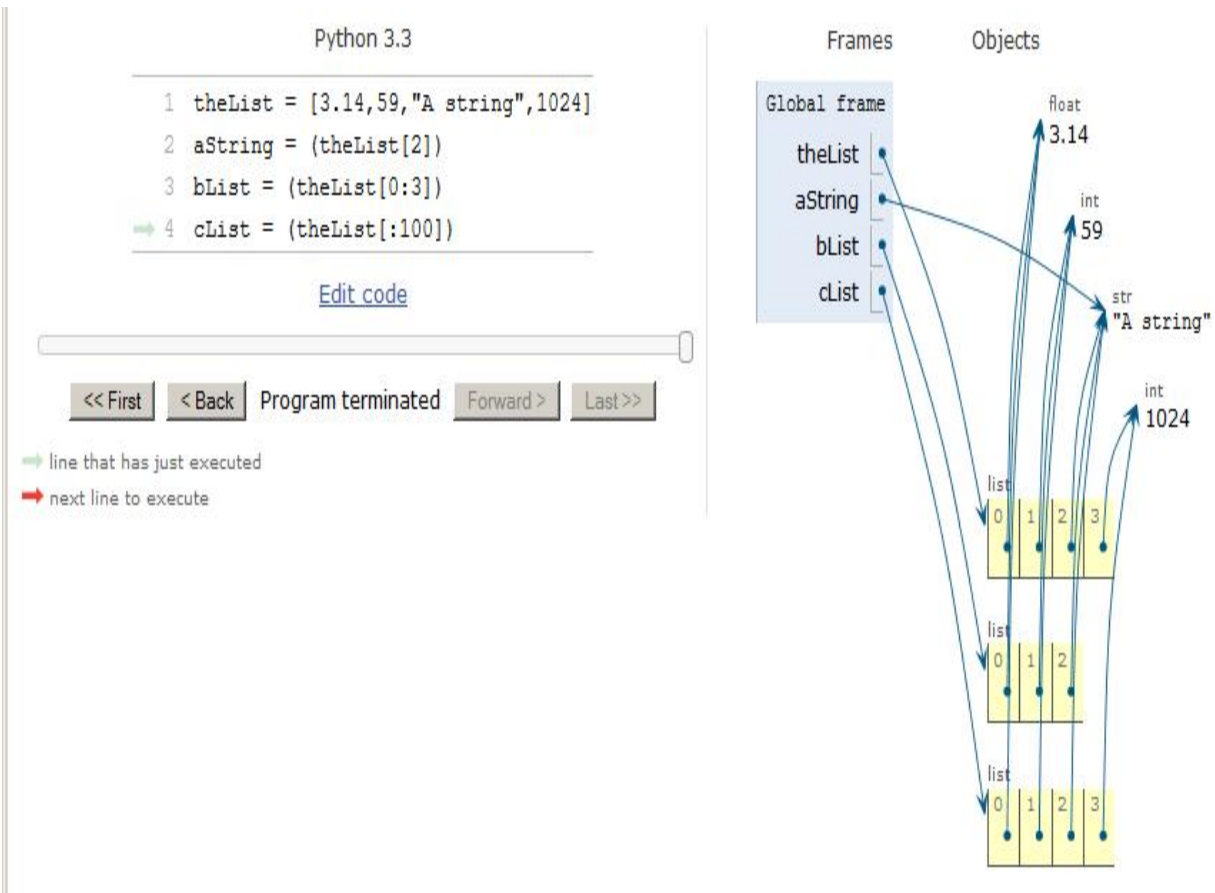
```
Print index value 2
A string
Print a short slice
[3.14, 59, 'A string']
Print the entire list
[3.14, 59, 'A string', 1024]
```

Visualize sliced lists

Create new lists by slicing an existing list

[Figure 2](#) shows the output from the [code visualizer](#) for code similar to the code in [Listing 4](#). Instead of accessing the list elements and printing them on the fly as in [Listing 4](#), the code in [Figure 2](#) accesses the list elements and saves references to them so that we can see what the resulting lists look like.

Figure 2. Visualize sliced lists.



The diagram on the right in [Figure 2](#) shows the state of the program in memory following the execution of all the code in the code box on the left. As you can see, there are now three separate lists:

- the original list pointed to by the variable named **theList** and
- two new lists pointed to by **bList** and **cList** .

The two new lists were created by slicing the original list. The elements in the lists all point to the **float** , **int** , and **str** objects that were used to populate the original list.

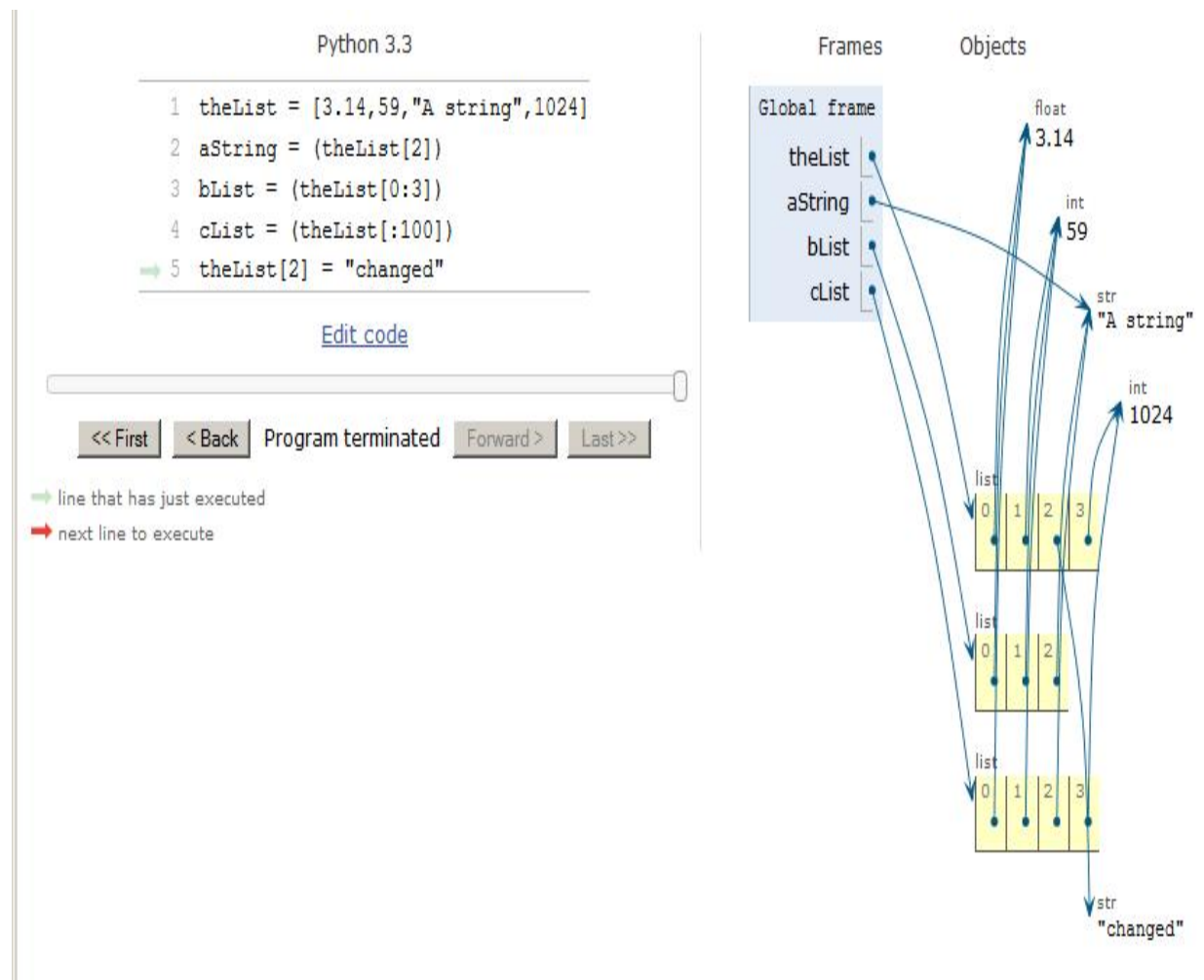
[Figure 2](#) illustrates a very important concept. Three of the variables shown in the light blue box point to objects of type list. The elements that populate those list objects point to other objects of the types **float** , **int** , and **str** (*This concept is often called indirection.*)

Since the two bottom lists were created as slices of the original (*top*) list, they all point to the same objects at this stage in the execution of the program.

Change an item in the original list

[Figure 3](#) shows what happens when one of the elements of the original list object is modified to cause it to point to a different object. (Note the arrow that now points from element 2 in the original list object down the screen to a new *str* object containing the word "**changed**".)

Figure 3. Change an item in the original list.



The important thing to note here is that the pointers in element 2 in the other two list objects did not follow suite. In other words, they still point to the original **str** object containing the words "**A string**" and they do not point to the new **str** object containing the word "**changed**". Thus, even though the two new list objects were created as slices from the original list object, once they are created, they are not dependent on the contents of the original list object. Subsequent changes in the contents of the original list object are not reflected in the contents of the two new list objects.

Lists can be concatenated

Lists can be concatenated using the + operator.

A concatenation program

The Python program shown in [Listing 5](#) creates two lists and prints them both. Then it concatenates the two lists and prints the concatenated version.

Listing 5 . Lists can be concatenated.

Listing 5 . Lists can be concatenated.

```
# Illustrates concatenating lists
#
#-----
print("Create two lists")
listA = [3.14,59,"A string",1024]
listB = [2,4,6,16]
print("Print listA")
print(listA)
print("Print listB")
print(listB)
print("Concatenate the lists")
listC = listA + listB
print("Print concatenated list")
print(listC)
```

Concatenation program output

The output from this program is shown in [Figure 4](#). As you can see, the concatenated list contains the elements of both of the individual lists.

Figure 4 . Concatenation program output.

Figure 4 . Concatenation program output.

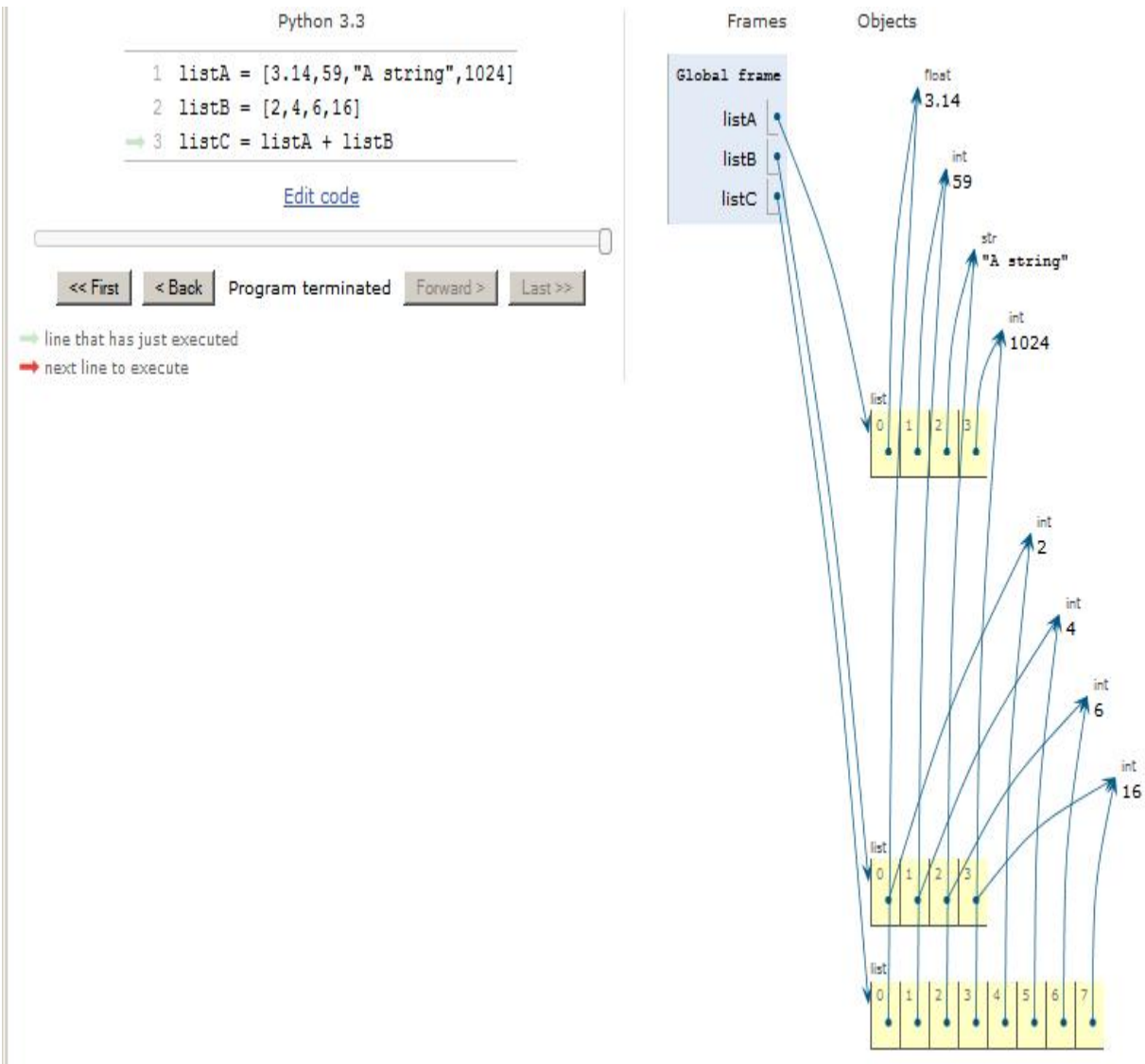
```
Create two lists
Print listA
[3.14, 59, 'A string', 1024]
Print listB
[2, 4, 6, 16]
Concatenate the lists
Print concatenated list
[3.14, 59, 'A string', 1024, 2, 4, 6, 16]
```

Visualize concatenated lists

Create a new list by concatenating two existing lists

The diagram in [Figure 5](#) visualizes the state of the program memory after the three statements in the code box have been executed. This code creates two list objects and then concatenates them into a third **list** object pointed to by the variable named **listC** .

Figure 5. Visualize concatenated lists.

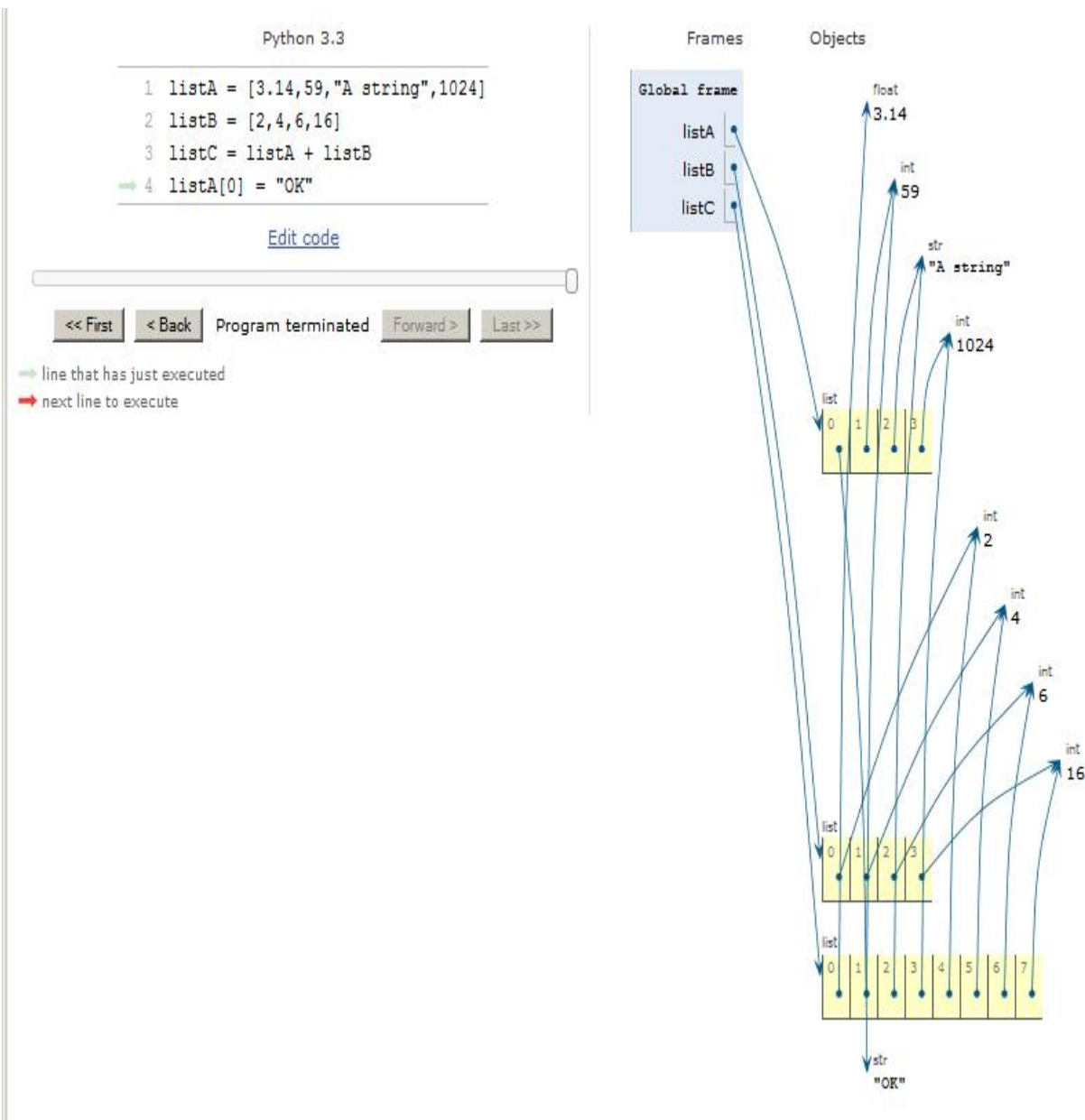


As you can see in [Figure 5](#), the elements in the new list object point to the objects originally pointed to by the elements in the two original list objects. As you can also see, the order of the elements in the two parts of the new list object preserve the order of the elements in the two original list objects.

Change an element in an original list

The code in [Figure 6](#) was updated with one additional statement that causes element 0 in one of the original list objects to point to a different object of type **str** containing " **OK** ".

Figure 6. Change an element in an original list.



However, element 0 in the concatenated list object pointed to by **listC** continues to point to the object of type **float** containing 3.14. The important thing to note here is that the pointer in element 0 in the new list object did not follow suite when there was a change in one of the elements in one of the original list objects. It still points to the original **float** object containing 3.14 and it does not point to the new **str** object containing "OK" . Therefore, even though the new list object was created by concatenating

two original list objects, once created, it is not dependent on the future contents of the original list objects. Subsequent changes in the contents of the original list objects are not reflected in the contents of the new list object.

Lists are mutable

Unlike strings, the values in a list can be modified after the list is created.

The Python program shown in [Listing 6](#) creates and prints a list. Then it uses a subscription to modify and print the list three times.

Listing 6 . Lists are mutable.

Listing 6 . Lists are mutable.

```
# Illustrates mutating lists
#
#-----
print("Create and print a list")
listA = [3.14,59,"A string",1024]
print(listA)

print("Modify the list")
listA[2] = "New string"
print("Print the modified list")
print(listA)

print("Modify the list again")
listA[3] = listA[3] * 2
print("Print the modified list")
print(listA)

print("Modify the list again")
listA[2] = 0.99999
print("Print the modified list")
print(listA)
```

Replace a string with another string

The first modification replaces an existing string in the list with a new string.

Multiply an integer element by two

The second modification multiplies an integer value in the list by a factor of two.

Replace a string by a float

The third modification replaces a string in the list by a float value of 0.99999.

List modification program output

The output from the program is shown in [Figure 7](#).

Figure 7 . List modification program output.

```
Create and print a list
[3.14, 59, 'A string', 1024]
Modify the list
Print the modified list
[3.14, 59, 'New string', 1024]
Modify the list again
Print the modified list
[3.14, 59, 'New string', 2048]
Modify the list again
Print the modified list
[3.14, 59, 0.99999, 2048]
```

More to come

There is a lot more for you to learn about lists that is not included in this module. I will continue this discussion of lists, including more sample programs, in a future module.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1080-Lists Part 1
- File: Itse1359-1080.htm
- Published: 10/15/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1080r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1080-Lists Part 1.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#)
- [Listing index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1080-Lists Part 1* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? A *subscription* in Python is how you go about getting magazines delivered to your mailbox.

Go to [answer 1](#)

Question 2

Name three types of *sequence* objects.

Go to [answer 2](#)

Question 3

Given the following nomenclature

primary "[" expression_list "]"

what is the requirement for the *primary* ?

Go to [answer 3](#)

Question 4

If the *primary* is a *sequence* , what must be the type of expression_list?

Go to [answer 4](#)

Question 5

True or False? If the primary is a *sequence* , the `expression_list` must evaluate to a positive integer or to a slice containing positive integers.

Go to [answer 5](#)

Question 6

Which item in the sequence is selected for an index value of -1?

Go to [answer 6](#)

Question 7

True or False? Just like C, C++, and Java, Python supports a *character* type.

Go to [answer 7](#)

Question 8

What is another name for a *mapping* ?

Go to [answer 8](#)

Question 9

What is a *mutable* sequence?

Go to [answer 9](#)

Question 10

True or False? A string is a sequence.

Go to [answer 10](#)

Question 11

True or False? A string is a mutable sequence.

Go to [answer 11](#)

Question 12

What kinds of items can be placed in a *list* ?

Go to [answer 12](#)

Question 13

How are *lists* formed from a syntax viewpoint?

Go to [answer 13](#)

Question 14

Show how to create a simple list using program code.

Go to [answer 14](#)

Question 15

Show how to access an item in a list using a subscription.

Go to [answer 15](#)

Question 16

Show how to access a slice from a list.

Go to [answer 16](#)

Question 17

Show how to concatenate two lists.

Go to [answer 17](#)

Question 18

Show how to modify an item in a list using a subscription.

Go to [answer 18](#)

Listing index

- [Listing 1](#). Answer 14.
- [Listing 2](#). Answer 15.
- [Listing 3](#). Answer 16.
- [Listing 4](#). Answer 17.
- [Listing 5](#). Answer 18.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 18

[Listing 5](#) shows how to modify an item in a list using a subscription.

Listing 5 . Answer 18.

```
listA = [3.14,59,"A string",1024]  
listA[2] = "New string"
```

Go back to [Question 18](#)

Answer 17

[Listing 4](#) shows how to concatenate two lists.

Listing 4 . Answer 17.

```
listA = [3.14,59,"A string",1024]  
listB = [2,4,6,16]  
listC = listA + listB
```

Go back to [Question 17](#)

Answer 16

[Listing 3](#) shows how to access a slice from a list.

Listing 3 . Answer 16.

```
print(theList[0:3])
```

Go back to [Question 16](#)

Answer 15

[Listing 2](#) shows how to access an item in a list using a subscription.

Listing 2 . Answer 15.

Listing 2 . Answer 15.

```
print(theList[2])
```

Go back to [Question 15](#)

Answer 14

[Listing 1](#) shows a simple list.

Listing 1 . Answer 14.

```
theList = [3.14,59,"A string",1024]
```

Go back to [Question 14](#)

Answer 13

Lists are formed by placing a comma-separated sequence of expressions in square brackets.

Go back to [Question 13](#)

Answer 12

The items of a *list* are arbitrary Python objects.

Go back to [Question 12](#)

Answer 11

False. The characters in a string cannot be modified after the string is created.

Go back to [Question 11](#)

Answer 10

True. A string is a sequence.

Go back to [Question 10](#)

Answer 9

Mutable sequences can be changed after they are created.

Go back to [Question 9](#)

Answer 8

A dictionary.

Go back to [Question 8](#)

Answer 7

False. There is no character type in Python. Rather, a string's items are characters. A character is not a separate data type but a string containing exactly one character.

Go back to [Question 7](#)

Answer 6

The last item in the sequence is selected for an index value of -1.

Go back to [Question 6](#)

Answer 5

False. The integers may be positive or negative.

Go back to [Question 5](#)

Answer 4

If the primary is a *sequence*, the `expression_list` must evaluate to an integer or a slice.

Go back to [Question 4](#)

Answer 3

The *primary* must evaluate to an object of a *sequence* or *mapping* type.

Go back to [Question 3](#)

Answer 2

string, tuple and list and possibly range object

Go back to [Question 2](#)

Answer 1

False in the Python context. In Python, A *subscription* selects an item of a *sequence* object.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1080r-Review
- File: Itse1359-1080r.htm
- Published: 10/15/14
- Revised: 02/21/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090pa-Preview-01-Lists Part 2

This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output for replacing a slice in a list](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Program output for replacing a slice in a list.

Listings

- [Listing 1](#). Program code template for replacing a slice in a list.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output for replacing a slice in a list

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for replacing a slice in a list.

```
1. [100, 200, 300, 400, 500]
2. 5
3. [100, 2, 4, 8, 16, 32, 64, 500]
4. 8
5. end
```

Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `???` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for replacing a slice in a list.

Listing 1. Program code template for replacing a slice in a list.

```
listA = [100,200,300,400,500]
"""Print the list as shown on Line 1."""
#??
"""Print length of list shown on Line 2."""
#??
"""Modify and print the list as shown on Line
3."""
#??
"""Print length of list shown on Line 4."""
#??

print("5.", "end")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1090-Lists Part 2](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090pa-Preview-01-Lists Part 2
- File: Itse1359-1090pa.htm
- Published: 03/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090pb-Preview-02-Lists Part 2

This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output for replacing an element with a list](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Program output.

Listings

- [Listing 1](#). Program code template.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output for replacing an element with a list

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for replacing an element with a list.

```
1. [100, 200, 300, 400, 500]
2. 5
3. [100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
4. 5
4. end
```

Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `???` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for replacing an element with a list.

Listing 1. Program code template for replacing an element with a list.

```
listA = [100,200,300,400,500]
"""Print the list as shown on Line 1."""
#??
"""Print length of list shown on Line 2."""
#??
"""Modify and print the list as shown on Line
3."""
#??
"""Print length of list shown on Line 4."""
#??

print("4.", "end")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1090-Lists Part 2](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090pb-Preview-02-Lists Part 2
- File: Itse1359-1090pb.htm

- Published: 03/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090pc-Preview-03-Lists Part 2

This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output for extracting elements from a nested list](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Program output for extracting elements from a nested list.

Listings

- [Listing 1](#). Program code template for extracting elements from a nested list.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output for extracting elements from a nested list

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for extracting elements from a nested list.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
Extract and display each
  element in the list
100
200
[2, 4, 8, 16, 32, 64]
400
500
1. 2
2. 4
3. 8
4. 16
5. 32
6. 64
7. End
```


Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `###` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for extracting elements from a nested list.

Listing 1. Program code template for extracting elements from a nested list.

```
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace an element")
listA[2] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
print("Extract and display each")
print(" element in the list")
print(listA[0])
print(listA[1])
print(listA[2])
print(listA[3])
print(listA[4])

"""Print the six output lines numbered 1.
through 6."""
#??
print("7. End")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1090-Lists Part 2](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090pc-Preview-03-Lists Part 2
- File: Itse1359-1090pc.htm
- Published: 03/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090pd-Preview-04-Lists Part 2

This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output for more nested elements](#).
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Program output for more nested elements.

Listings

- [Listing 1](#). Program code template for more nested elements.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output for more nested elements

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for more nested elements.

```
Create and print a list with 3 nested elements
[[2, 4], [8, 16, 32], [64, 128, 256, 512]]
1. Number of elements is:
2. 3
3. Length of Element 0 is
4. 2
5. Length of Element 1 is
6. 3
7. Length of Element 2 is
8. 4
9. End
```

Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `###` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for more nested elements.

```
print("Create and print a list\  
with 3 nested elements")  
listA = [[2,4],[8,16,32],[64,128,256,512]]  
print(listA)  
"""Print the output lines numbered 1 through  
8."""  
#??  
print("9. End")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1090-Lists Part 2](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090pd-Preview-04-Lists Part 2
- File: Itse1359-1090pd.htm
- Published: 03/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090pe-Preview-05-Lists Part 2

This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Program output for a three-dimensional array program](#)
 - [Instructions](#)
 - [Answers in the back of the book](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module provides a preview of code that will be explained in more detail in the module titled [Itse1359-1090-Lists Part 2](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Program output for a three-dimensional array program.

Listings

- [Listing 1](#). Program code template for a three-dimensional array program.

General background information

This module provides a preview of code that will be explained in more detail in a future module. The purpose of this module is to provide an opportunity for you to work through and to think about the code on your own before reading the explanation. This should give you a clear assessment of what you already know and what you don't yet understand. The hope is that by working through and thinking about the code before reading the explanation, you will be better prepared to retain the knowledge later when you do read the explanation.

This approach is based loosely on the concept of [Intentional Programming](#) as incorporated in the instructor-led Java programming courseware that is available from [TKP](#).

Discussion and sample code

Program output for a three-dimensional array program

You will be writing code based on instructions provided in comments. Once this program is completed, it should produce the output shown in [Figure 1](#).

Figure 1. Program output for a three-dimensional array program.

```
Create and print athree-dimensional array list
[[[1], [2]], [[3], [4]], [[5], [6]], [[7],
[8]]]
Print each element
1. [1]
2. [2]
3. [3]
4. [4]
5. [5]
6. [6]
7. [7]
8. [8]
9. End
```

Instructions

Copy the code template from [Listing 1](#) into your favorite Python text editor or IDE. Then make the modifications described in the comments by replacing each instance of `???` with one or more statements. Feel free to run the program at any point in the process in order to compare your output with the required output shown in [Figure 1](#).

Listing 1. Program code template for a three-dimensional array program.

```
print("Create and print a\
three-dimensional array list")
listA = [[[[1],[2]],[[3],[4]]],[[5],[6]],
[[7],[8]]]
print(listA)
print("Print each element")
#??
print("9.", "End")
```

Answers in the back of the book

If you get stuck and are unable to successfully complete one of the modifications, you will find the correct code in the module titled [Itse1359-1090-Lists Part 2](#) except that the code in that module does not produce the line numbers shown in [Figure 1](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090pe-Preview-05-Lists Part 2
- File: Itse1359-1090pe.htm
- Published: 03/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090-Lists Part 2

This module expands on the previous module on lists by teaching you other ways to manipulate lists.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preface](#)
- [Introduction](#)
 - [Manipulating lists](#)
 - [Other ways to manipulate lists](#)
- [Some sample programs](#)
 - [Replacing a slice](#)
 - [Can change the length of the list](#)
 - [The function named len\(\)](#)
 - [Replaces a three-element slice with a six-element list](#)
 - [Length of the list is increased by 3](#)
 - [Program output](#)
 - [Replacing an element with a list](#)
 - [Behavior is different from above](#)
 - [Produces nested lists](#)
 - [Length is unchanged](#)
 - [Program output from replacing an element with a list](#)
 - [One element is itself a list](#)
 - [A visualization](#)

- [Extracting elements from a nested list](#)
 - [Display the nested list combination](#)
 - [Double-square-bracket notation](#)
 - [Program output from extracting elements from a nested list](#)
- [More on nested elements](#)
 - [Create a list of lists](#)
 - [Inner lists are different lengths](#)
 - [Program behavior and output](#)
 - [Getting the length of a nested list](#)
- [Arrays -- not for beginners](#)
 - [Python lists are more powerful](#)
 - [Sub-arrays can be different sizes](#)
 - [Types can also be different](#)
- [A three-dimensional array_program](#)
 - [Triple-square-bracket notation](#)
 - [Visualization of a three-dimensional array](#)
 - [Program behavior](#)
 - [Output from a three-dimensional array_program](#)
- [More information on lists](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It expands on the previous module titled [Itse1359-1080-Lists Part 1](#) by teaching you some other ways to manipulate lists.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Program output.
- [Figure 2](#). Program output from replacing an element with a list.
- [Figure 3](#). Visualization
- [Figure 4](#). Program output from extracting elements from a nested list.
- [Figure 5](#). Output from more nested elements.
- [Figure 6](#). Visualization of a three-dimensional array.
- [Figure 7](#). Output from a three-dimensional array program.

Listings

- [Listing 1](#). Replace a slice in a list.
- [Listing 2](#). Replacing an element with a list.
- [Listing 3](#). Extracting elements from a nested list.
- [Listing 4](#). More nested elements.
- [Listing 5](#). A three-dimensional array program.

Introduction

A previous module titled [Itse1359-1080-Lists Part 1](#) introduced you to lists and some other structures as well.

It also introduced you to

- subscriptions
- sequences
- mutable sequences
- mappings
- slicing, and
- tuples.

Manipulating lists

That module showed you some of the ways that you can manipulate lists. The discussion was illustrated using sample programs and visualizations.

Other ways to manipulate lists

This module carries that discussion forward by using sample programs to teach you other ways to manipulate lists.

Some sample programs

Replacing a slice

You can replace a slice in a list with the elements from another list through assignment.

Can change the length of the list

Note that this operation can change the length of the original list.

The script in [Listing 1](#) replaces a slice in a list with the elements from another list.

Listing 1 . Replace a slice in a list.

```
# Illustrates replacing a slice in a list
#
#-----
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace a slice")
listA[1:4] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
```

The function named len()

This program also illustrates the use of the function named **len()** to get and print the length of the list.

Replaces a three-element slice with a six-element list

In this program, a slice of an original five-element list, consisting of the elements from 1 through 3 inclusive, is replaced by the elements of a new list consisting of six new elements.

Length of the list is increased by 3

Since three existing elements are replaced by six new elements, the overall length of the list is increased by three elements.

Program output

The output from this program is shown in [Figure 1](#).

Figure 1 . Program output.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace a slice
Print the modified list
[100, 2, 4, 8, 16, 32, 64, 500]
Modified length is:
8
```

As you can see in [Figure 1](#), the six new elements replaced the three original elements to increase the length of the list from 5 to 8 elements.

Replacing an element with a list

It is also possible to replace an element in an existing list with a new list, as illustrated by the program in [Listing 2](#).

Behavior is different from above

In this case, the behavior is different from that shown above where a slice from the original list was replaced with the elements from a different list (*even though the right operand of the assignment operator is the same in both cases*) .

Produces nested lists

When a single element is replaced by a list, the result is that *a new list is nested inside the original list* .

Length is unchanged

It is also interesting to note that the length of the list is unchanged by this operation since the list that replaces the element is itself considered to be a single element. Therefore, the number of elements is not changed.

This is illustrated in [Listing 2](#), where the element at index 2 of an original list is replaced with a new list having six elements.

Listing 2 . Replacing an element with a list.

Listing 2 . Replacing an element with a list.

```
# Illustrates replacing an
# element with a slice
#
#-----
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace an element")
listA[2] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
```

Program output from replacing an element with a list

The output from this program is shown in [Figure 2](#).

Figure 2 . Program output from replacing an element with a list.

Figure 2 . Program output from replacing an element with a list.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
```

One element is itself a list

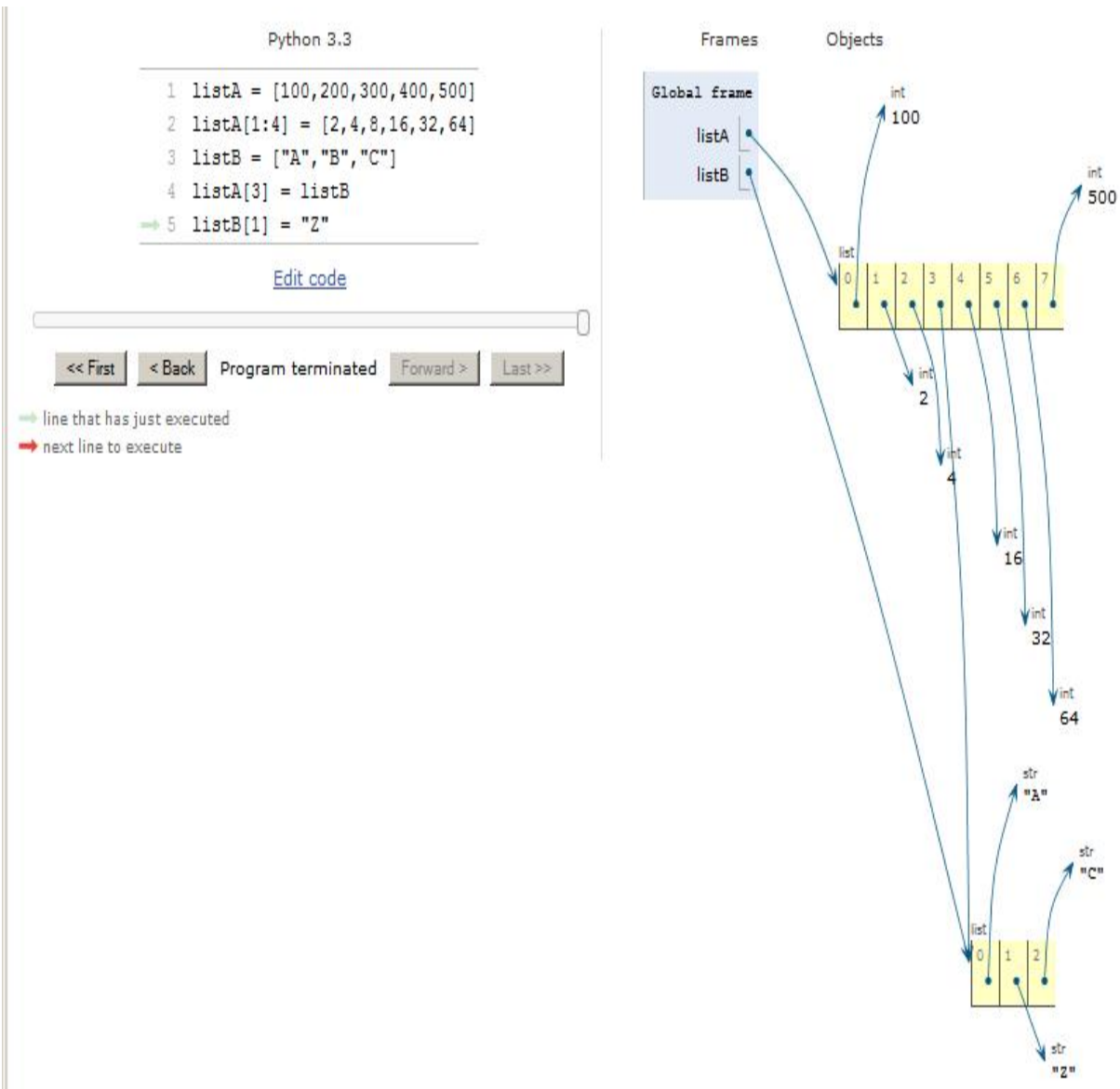
As you can see from [Figure 2](#), the result is that one of the elements in the original five-element list is replaced by a new list containing six elements. However, the length of the list is unchanged.

Again, it is important to note that this results in one list being nested inside of another list.

A visualization

The code visualization shown in [Figure 3](#) summarizes the concepts that were explained in the previous two sections. This image was produced by entering the five statements shown in the [code visualizer's](#) code window and stepping through all five instructions.

Figure 3. Visualization.



I recommend that you use the [code visualizer](#) to step through this code and observe the state of the program in memory as each instruction is executed. Try to correlate the behavior of this program with what you learned in the sections above titled [Replacing a slice](#) and [Replacing an element with a list](#).

Extracting elements from a nested list

Now I am going to illustrate the syntax for extracting elements from a nested list using *pairs of matching square brackets* . [Listing 3](#) is an expansion of [Listing 2](#).

Listing 3 . Extracting elements from a nested list.

```
# Illustrates extracting a
# list element and extracting
# elements from a nested list
#
#-----
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace an element")
listA[2] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
print("Extract and display each")
print(" element in the list")
print(listA[0])
print(listA[1])
print(listA[2])
print(listA[3])
print(listA[4])
```


Listing 3 . Extracting elements from a nested list.

```
print("Extract and display each")
print(" element in nested list")
print(listA[2][0])
print(listA[2][1])
print(listA[2][2])
print(listA[2][3])
print(listA[2][4])
print(listA[2][5])
```

Display the nested list combination

After nesting a list as element 2 in another list, the program displays the value of each of the elements of the original list. When element 2 is displayed, it can be seen to be another list.

Double-square-bracket notation

Then the program uses double-square-bracket notation (*listA[2][4]*) to extract and display each of the elements in the nested inner list that comprises element 2 of the outer list.

Program output from extracting elements from a nested list

The output from this program is shown in [Figure 4](#).

Figure 4 . Program output from extracting elements from a nested list.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
Extract and display each
element in the list
100
200
[2, 4, 8, 16, 32, 64]
400
500
Extract and display each
element in nested list
2
4
8
16
32
64
```

More on nested elements

The program in [Listing 4](#) illustrates some additional aspects of nested elements.

Listing 4 . More nested elements.

```
# Illustrates more nested elements
#
#-----
print("Create and print a list\
with 3 nested elements")
listA = [[2,4],[8,16,32],[64,128,256,512]]
print(listA)
print("Number of elements is:")
print(len(listA))
print("Length of Element 0 is")
print(len(listA[0]))
print("Length of Element 1 is")
print(len(listA[1]))
print("Length of Element 2 is")
print(len(listA[2]))
```

Create a list of lists

This program defines a three-element list containing three nested lists.

In other words, each of the elements in the outer list is itself a list.

Inner lists are different lengths

Furthermore, the lengths of the inner nested lists are not the same. The lengths of the inner nested lists are 2, 3, and 4 elements each, respectively.

Program behavior and output

The output from the program is shown in [Figure 5](#). The program displays the entire list, and then gets and displays the lengths of each of the nested lists.

Figure 5 . Output from more nested elements.

```
Create and print a list with 3 nested elements
[[2, 4], [8, 16, 32], [64, 128, 256, 512]]
Number of elements is:
3
Length of Element 0 is
2
Length of Element 1 is
3
Length of Element 2 is
4
```

Getting the length of a nested list

Note in particular the syntax used to pass a parameter to the **len()** method in order to get the length of a nested list (**len(listA[1])**) .

Arrays -- not for beginners

If you are a beginning programmer, you may want to skip this section. If you are an experienced programmer, you may have observed that a Python

lists bear a striking resemblance to arrays in other programming environments.

Python lists are more powerful

However, Python lists are more powerful than the arrays I am aware of in other programming environments, including Java.

Sub-arrays can be different sizes

For example as in Java, when a Python list is used to construct a multidimensional array, the sub arrays don't have to be of the same size.

Types can also be different

However, unlike Java, the elements in the array don't even have to be of the same type (*granted that the elements in a Java array can be of different types so long as there is an inheritance or Interface relationship among them*).

A three-dimensional array program

The program in [Listing 5](#) might represent what an experienced programmer would consider to be a three-dimensional array of integer data in some other programming environment.

Listing 5 . A three-dimensional array program.

Listing 5 . A three-dimensional array program.

```
# Illustrates a three-dimensional array list
#
#-----
-----
print("Create and print a\
three-dimensional array list")
listA = [[[[1],[2]],[[3],[4]]],[[5],[6]],
[[7],[8]]]
print(listA)
print("Print each element")
print(listA[0][0][0])
print(listA[0][0][1])
print(listA[0][1][0])
print(listA[0][1][1])
print(listA[1][0][0])
print(listA[1][0][1])
print(listA[1][1][0])
print(listA[1][1][1])
```

Triple-square-bracket notation

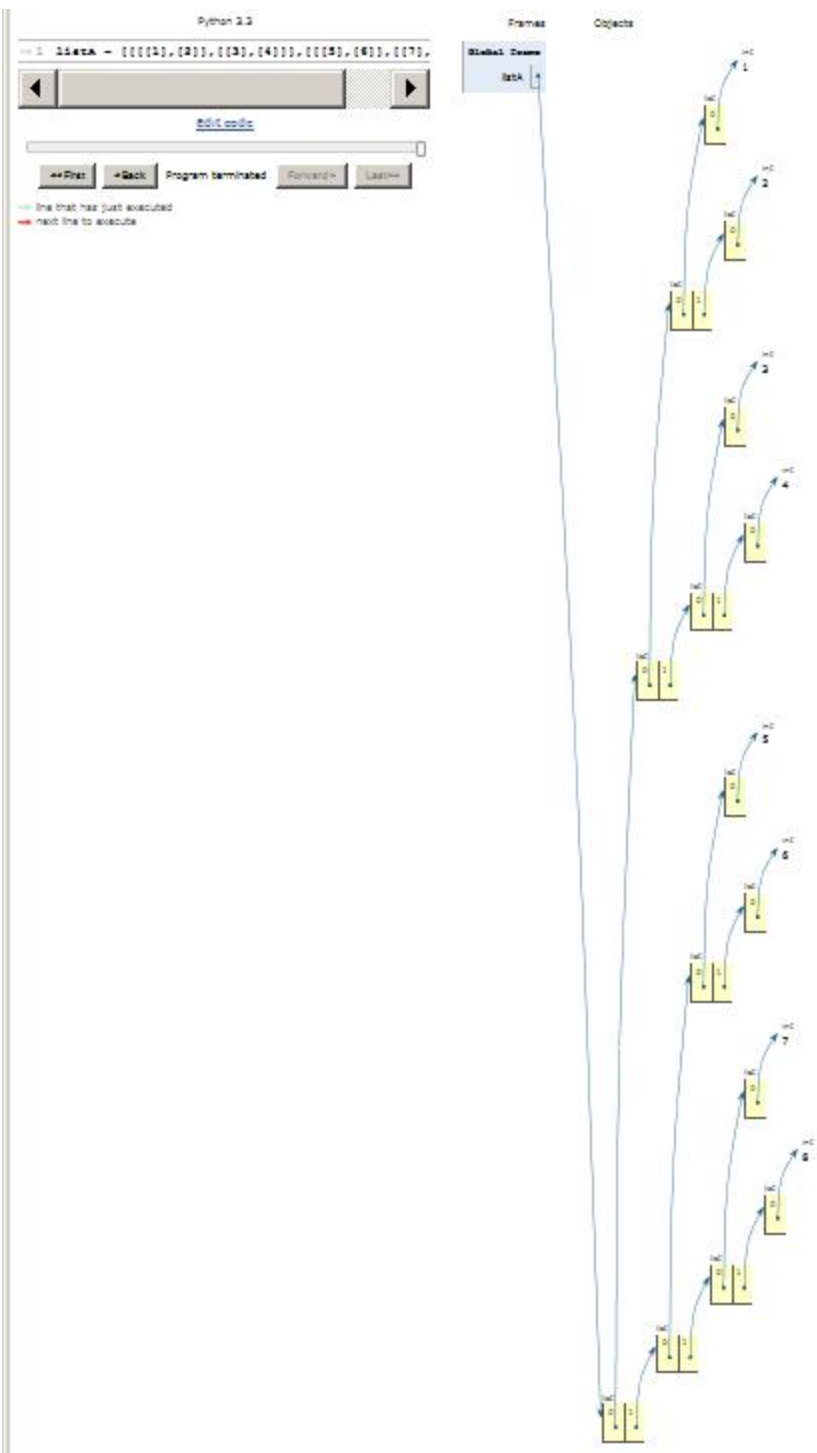
Pay particular attention to the triple square bracket notation that is used to access and print each element in the array. This is very similar to the syntax in other programming languages.

Visualization of a three-dimensional array

[Figure 6](#) shows a [visualization](#) of the three-dimensional array pointed to by the variable named **listA** in [Listing 5](#). In programming, we deal with arrays

having one, two, three or more dimensions. As you can see, even a three-dimensional array is a relatively complex structure.

Figure 6. Visualization of a three-dimensional array.



Program behavior

This program

- Creates and populates the list that represents a three-dimensional array.
- Displays the entire array as a set of nested lists.
- Displays the contents of each element in the array.

Output from a three-dimensional array program

The output from the program is shown in [Figure 7](#). The triple-square-bracket notation in the program of [Listing 5](#) is essentially the same notation that would be used to access the individual elements in a three-dimensional array in C, C++, or Java.

Figure 7 . Output from a three-dimensional array program.

Figure 7 . Output from a three-dimensional array program.

```
Create and print a three-dimensional array  
list  
[[[1], [2]], [[3], [4]], [[5], [6]], [[7],  
[8]]]  
Print each element  
[1]  
[2]  
[3]  
[4]  
[5]  
[6]  
[7]  
[8]
```

More information on lists

There is quite a lot more that you will need to learn about lists. However, I will defer that discussion for a future module where I discuss the use of a list as a *data structure* or a *container* .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090-Lists Part 2
- File: Itse1359-1090.htm
- Published: 10/15/14

- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1090-Lists Part 2.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#)
- [Image index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1090-Lists Part 2* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

Show how to replace a slice of a list with the elements from a new list.

Go to [answer 1](#)

Question 2

Show how to replace an element in a list with a nested list.

Go to [answer 2](#)

Question 3

Show how to extract the elements from one list that is nested in another list.

Go to [answer 3](#)

Question 4

Show how to create a list of nested lists where the nested lists have different lengths.

Go to [answer 4](#)

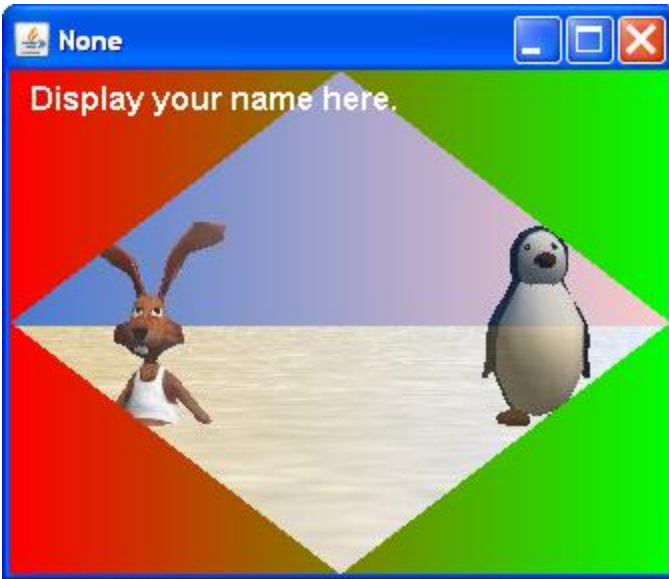
Image index

- [Image 1](#). Answer 1.
- [Image 2](#). Answer 2.
- [Image 3](#). Answer 3.

- [Image 4](#). Answer 4.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 4

A solution is shown in Image 4.

Image 4 . Answer 4.

```
listA = [[2,4],[8,16,32],[64,128,256,512]]
```

Go back to [Question 4](#)

Answer 3

A solution is shown in Image 3.

Image 3 . Answer 3.

Image 3 . Answer 3.

```
listA = [100,200,300,400,500]
listA[2] = [2,4,8,16,32,64]
print(listA[2][0])
print(listA[2][1])
print(listA[2][2])
print(listA[2][3])
print(listA[2][4])
print(listA[2][5])
```

Go back to [Question 3](#)

Answer 2

A solution is shown in Image 2.

Image 2 . Answer 2.

```
listA = [100,200,300,400,500]
listA[2] = [2,4,8,16,32,64]
```

Go back to [Question 2](#)

Answer 1

A solution is shown in Image 1.

Image 1 . Answer 1.

```
listA = [100,200,300,400,500]
listA[1:4] = [2,4,8,16,32,64]
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090r-Review
- File: Itse1359-1090r.htm
- Published: 10/15/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1100-Indexing and Slicing Tuples

This module and several modules that follow will use sample programs to show you a variety of ways to manipulate and use tuples.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
- [What is a tuple?](#)
 - [Description of a tuple](#)
 - [An array of references](#)
 - [What can you do with a tuple?](#)
 - [Why do tuples exist?](#)
- [A sample program](#)
 - [Indexing and slicing tuples](#)
 - [The program output](#)
 - [Tuple syntax](#)
 - [Indexing items in a tuple](#)
 - [Tuples can be sliced](#)
- [Complete program listing](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin

Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Visualization of a tuple.
- [Figure 2](#). Output from code in Listing 6.

Listings

- [Listing 1](#). Beginning of the program.
- [Listing 2](#). Tuple code with parentheses removed.
- [Listing 3](#). Indexing items in a tuple.
- [Listing 4](#). Printing a short slice from a tuple.
- [Listing 5](#). Print the entire tuple.
- [Listing 6](#). Complete program listing.

Introduction

Previous modules have introduced you to lists, subscriptions, sequences, mutable sequences, mappings, slicings, and have mentioned tuples.

Those modules showed you some of the ways that you can manipulate lists. The discussion was illustrated using sample programs.

The introduction to tuples in previous modules was very brief. This and several future modules will use sample programs to show you a variety of ways to manipulate and use tuples.

What is a tuple ?

Description of a tuple

As a practical matter, a tuple is like a list whose values cannot be modified. In other words, a tuple is *immutable* .

According to Lutz and Ascher, Learning Python from O'Reilly, tuples are "*Ordered collections of arbitrary objects.*"

Again according to Lutz and Ascher, "*They work exactly like lists, except that tuples can't be changed in place (they're immutable)...*"

Unlike lists, however, tuples don't use square brackets for containment. Rather, they are normally written as a sequence of items contained in parentheses.

Like a string or a list, a tuple is a *sequence* . Like a string (*but unlike a list*) , a tuple is an *immutable* sequence.

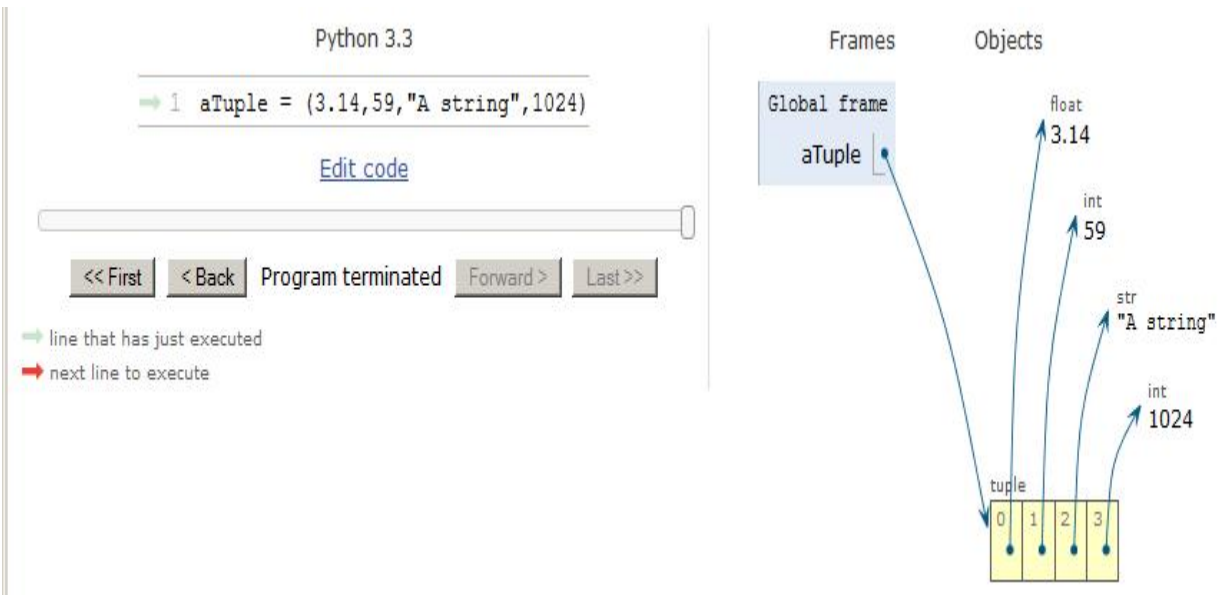
Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

An array of references

One way to think of a tuple is to consider it to be an array of references to other objects.

For example, [Figure 1](#) shows a [visualization](#) of the tuple from [Listing 6](#) . As you can see, each element in the tuple object at the bottom right contains a reference or a pointer to another object.

Figure 1. Visualization of a tuple.



While the tuple itself cannot be changed in place, the values contained in the objects that are referred to by the contents of the tuple can be changed (*assuming that those objects are mutable*) .

What can you do with a tuple ?

You can do just about anything with a tuple that you can do with a list, taking into account the fact that the tuple is *immutable* . Therefore, those list operations that change the value of a list in place cannot be performed on a tuple. For example, you cannot call the **append** method on a tuple as you can on a list.

As with strings and lists, items in a tuple are accessed using a numeric index. The first item in a tuple is at index value 0.

Why do tuples exist ?

Tuples provide some degree of integrity to the data stored in them. You can pass a tuple around through a program and be confident that its value can't

be accidentally changed. As mentioned earlier, however, the values stored in the items referred to in a tuple can be changed.

In addition, in a future module, you will see some sample programs that require the use of tuples.

A sample program

Indexing and slicing tuples

[Listing 1](#) shows the beginning of a Python program that first creates, and then manipulates a simple tuple.

Listing 1 . Beginning of the program.

```
# Illustrates indexing and slicing a simple
tuple
#-----
---

print("Create a simple tuple")
aTuple = (3.14,59,"A string",1024)
```

The remainder of this program will be shown and discussed as code fragments in subsequent Listings. The entire program is shown in [Listing 6](#) near the end of the module.

The program output

At this point, I am going to show you the output produced by executing the program so that you will have it available for reference during the discussion that follows. The output from the code in [Listing 6](#) is shown in [Figure 2](#).

Figure 2 . Output from code in Listing 6.

```
Create a simple tuple
Print index value 2
A string
Print a short slice
(3.14, 59, 'A string')
Print the entire tuple
(3.14, 59, 'A string', 1024)
```

Tuple syntax

From a syntax viewpoint, you create a tuple by placing a sequence of items inside a pair of enclosing parentheses and separating them by commas.

Note that the parentheses can be omitted when such omission will not lead to ambiguity.

The fragment in [Listing 1](#) creates a simple four-item tuple and assigns it to the variable named **aTuple** .

Note that the items in a tuple can be different types. This simple tuple contains a float, an integer, a string, and another integer.

In this case, the parentheses could be omitted from the tuple syntax, because such omission would not lead to ambiguity. [Listing 2](#) shows what this code fragment would look like if the parentheses were omitted.

Listing 2 . Tuple code with parentheses removed.

```
# Illustrates indexing and slicing a simple
tuple
#-----
---

print("Create a simple tuple")
aTuple = 3.14,59,"A string",1024
```

The code fragments in [Listing 1](#) and [Listing 2](#) are operationally identical and produce the same output, as shown in [Figure 2](#).

Indexing items in a tuple

The items in a tuple can be accessed using an index enclosed in square brackets as shown in [Listing 3](#). *(Earlier modules showed how to use an*

index in square brackets to access the items in a list.)

Listing 3 . Indexing items in a tuple.

```
print("Print index value 2")  
print(aTuple[2])
```

The third item in the tuple is accessed and printed in [Listing 3](#). *(Remember, index values begin with the value 0, so index value 2 points to the third item in the tuple.)*

The second and third lines of text in [Figure 2](#) were produced by the code in [Listing 3](#).

Tuples can be sliced

Tuples can be sliced just like lists and strings. This is illustrated by the code in [Listing 4](#).

Listing 4 . Printing a short slice from a tuple.

Listing 4 . Printing a short slice from a tuple.

```
print("Print a short slice")  
print(aTuple[0:3])
```

The code in [Listing 4](#) uses a slice to access and print the first three items in the tuple. *(Remember, a slice begins with the index shown by the first specified value and ends with the index whose value is one less than the second specified value.)*

The output produced by the code in [Listing 4](#) is shown by the fourth and fifth lines of text in [Figure 2](#).

Finally, the code fragment in [Listing 5](#) causes the entire tuple to be accessed and printed, as shown by the last two lines of text in [Figure 2](#).

Listing 5 . Print the entire tuple.

```
print("Print the entire tuple")  
print(aTuple[:100])
```

Complete program listing

A complete listing of the program is shown in [Listing 6](#).

Listing 6 . Complete program listing.

```
# Illustrates indexing and slicing a simple
tuple
#-----
---

print("Create a simple tuple")
aTuple = (3.14,59,"A string",1024)
print("Print index value 2")
print(aTuple[2])
print("Print a short slice")
print(aTuple[0:3])
print("Print the entire tuple")
print(aTuple[:100])
```

Run the program

I encourage you to copy the code from [Listing 6](#). Execute the code and confirm that you get the same results as those shown in [Figure 2](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1100-Indexing and Slicing Tuples

- File: Itse1359-1100.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1100r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1100-Indexing and Slicing Tuples.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#)
- [Figure index](#)
- [Listing index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1100-Indexing and Slicing Tuples* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? A tuple is like a list whose values cannot be modified. A tuple is *immutable* .

Go to [answer 1](#)

Question 2

True or False? A tuple is an ordered collections of arbitrary objects.

Go to [answer 2](#)

Question 3

True or False? Tuples work exactly like lists, except that tuples can't be changed in place (they're immutable).

Go to [answer 3](#)

Question 4

True or False? Tuples use square brackets for containment.

Go to [answer 4](#)

Question 5

True or False? A tuple is a *sequence* .

Go to [answer 5](#)

Question 6

True or False? Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

Go to [answer 6](#)

Question 7

True or False? Because a tuple is immutable, the values contained in the objects that are referred to by the contents of the tuple cannot be changed.

Go to [answer 7](#)

Question 8

True or False? Just like with a list, the **append** method can be called to add new items to a tuple.

Go to [answer 8](#)

Question 9

True or False? As with strings and lists, items in a tuple are accessed using a numeric index. The first item in a tuple is at index value 0.

Go to [answer 9](#)

Question 10

True or False? From a syntax viewpoint, you always create a tuple by placing a sequence of items inside a pair of enclosing parentheses and separating them by commas.

Go to [answer 10](#)

Question 11

True or False? The code in [Listing 1](#) produces the output shown in [Figure 1](#).

Listing 1 . Question 11 code.

```
aTuple = 3.14,59,"A string",1024  
print(aTuple)
```

Figure 1 . Question 11 possible output.

Figure 1 . Question 11 possible output.

3.14, 59, 'A string', 1024

Go to [answer 11](#)

Question 12

True or False? The code in [Listing 2](#) produces the output shown in [Figure 3](#) .

Listing 2 . Question 12 code.

```
# File 1359-1100r-12.py
#-----
aTuple = 3.14,59,"A string",1024
print(aTuple[4])
```

Figure 3 . Question 12 possible output.

Figure 3 . Question 12 possible output.

1024

Go to [answer 12](#)

Question 13

True or False? The code in [Listing 3](#) produces the output shown in [Figure 5](#) .

Listing 3 . Question 13 code.

```
aTuple = 3.14,59,"A string",1024  
print(aTuple[1:3])
```

Figure 5 . Question 13 possible output.

Figure 5 . Question 13 possible output.

(59, 'A string')

Go to [answer 13](#)

Figure index

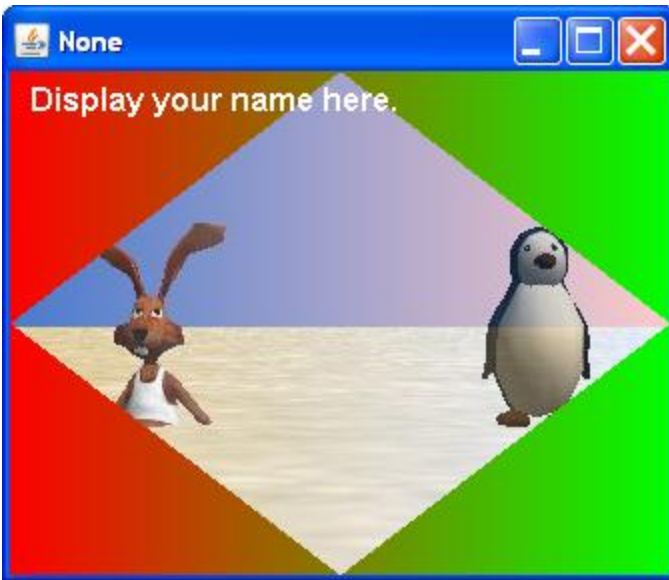
- [Figure 1](#). Question 11 possible output.
- [Figure 2](#). Answer 11 actual output.
- [Figure 3](#). Question 12 possible output.
- [Figure 4](#). Question 12 actual output.
- [Figure 5](#). Question 13 possible output.

Listing index

- [Listing 1](#). Question 11 code.
- [Listing 2](#). Question 12 code.
- [Listing 3](#). Question 13 code.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 13

True.

Go back to [Question 13](#)

Answer 12

False. The actual output is shown in [Figure 4](#). While it is possible to extract and print an element from a tuple using square-bracket indexing, tuple indices are zero based and the index shown in [Listing 2](#) is out of range.

Figure 4 . Question 12 actual output.

```
Traceback (most recent call last):  
  File "1359-1100r-12.py", line 4, in <module>  
    print(aTuple[4])  
IndexError: tuple index out of range
```

Go back to [Question 12](#)

Answer 11

False. Even though you can sometimes omit the parentheses when creating a tuple, those parentheses are restored when the tuple is printed as shown in [Figure 2](#).

Figure 2 . Answer 11 actual output.

```
(3.14, 59, 'A string', 1024)
```

Go back to [Question 11](#)

Answer 10

False. From a syntax viewpoint, you typically create a tuple by placing a sequence of items inside a pair of enclosing parentheses and separating them by commas.

However, and this is important, the parentheses can be omitted when such omission will not lead to ambiguity.

Go back to [Question 10](#)

Answer 9

True.

Go back to [Question 9](#)

Answer 8

False. You can do just about anything with a tuple that you can do with a list, taking into account the fact that the tuple is *immutable* . Therefore, those list operations that change the value of a list in place cannot be

performed on a tuple. For example, you cannot call the **append** method on a tuple as you can on a list.

Go back to [Question 8](#)

Answer 7

False. While a tuple itself cannot be changed in place, the values contained in the objects that are referred to by the contents of the tuple can be changed (*assuming that those objects are mutable*) .

Go back to [Question 7](#)

Answer 6

True.

Go back to [Question 6](#)

Answer 5

True.

Go back to [Question 5](#)

Answer 4

False. Tuples are normally written as a sequence of items contained in parentheses.

Go back to [Question 4](#)

Answer 3

True.

Go back to [Question 3](#)

Answer 2

True.

Go back to [Question 2](#)

Answer 1

True. A tuple is *immutable* .

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1100r-Review
- File: Itse1359-1100r.htm
- Published: 10/19/14
- Revised: 06/12/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1110-Nested Tuples

This module will expand your knowledge of tuples by teaching you about nesting tuples within other tuples.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
- [What is a tuple?](#)
- [Sample program](#)
 - [Create two tuples](#)
 - [Nesting the tuples](#)
 - [Get the length of the new tuple](#)
- [Complete program listing](#)
- [Visualization of the tuples](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the

Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Output from the code in Listing 1.
- [Figure 2](#). Output from the code in Listing 2.
- [Figure 3](#). Output from the code in Listing 3.
- [Figure 4](#). Output from the code in Listing 4.
- [Figure 5](#). Visualization of the tuples

Listings

- [Listing 1](#). Beginning of the program.
- [Listing 2](#). Nesting the tuples.
- [Listing 3](#). Get the length of the new tuple.
- [Listing 4](#). Complete program listing.

Introduction

Previous modules have introduced you to lists, subscriptions, sequences, mutable sequences, mappings, slicings, and tuples.

The earlier module titled [Itse1359-1100-Indexing and Slicing Tuples](#) showed you:

- How to create a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.

This module will expand your knowledge of tuples by teaching you about nesting tuples within other tuples.

What is a tuple ?

To briefly repeat part of what you learned in the earlier module, a tuple is like a list whose values cannot be modified. In other words, a tuple is immutable.

- Tuples are normally written as a sequence of items contained in (*optional*) matching parentheses.
- A tuple is an immutable sequence.
- Items in a tuple are accessed using a numeric index.

Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested inside of other tuples.

Sample program

[Listing 4](#) shows a complete listing of a Python program that

- Creates two tuples.
- Nests them in a third tuple.
- Determines the length (*number of items*) in the tuple containing the two nested tuples.
- Prints various items of information along the way.

A [visualization](#) of the tuples in this program is provided in [Figure 5](#).

Create two tuples

The beginning of the program is shown in [Listing 1](#).

Listing 1 . Beginning of the program.

```
# Illustrates creating and displaying nested  
tuples  
#-----  
-----  
  
print("Create/print one tuple")  
t1 = 1,2  
print(t1)  
print("Create/print another tuple")  
t2 = "a", "b"  
print(t2)
```

The code in [Listing 1](#) creates and displays the two tuples.

[Figure 1](#) shows the output produced by the code fragment in [Listing 1](#).

Figure 1 . Output from the code in Listing 1.

```
Create/print one tuple  
(1, 2)  
Create/print another tuple  
( 'a', 'b' )
```

The remaining parts of this program will be presented and discussed as code fragments in subsequent Listings. As mentioned above, a complete listing of the program is shown in [Listing 4](#).

A consolidated view of the output from the program is shown in [Figure 4](#).

Nesting the tuples

The code in [Listing 2](#) nests the two tuples, **t1** and **t2** , produced earlier, along with two strings, in a new tuple. The new tuple is assigned to the variable named **t3** .

Listing 2 . Nesting the tuples.

```
print("Create/print nested tuple")
t3 = "A",t1,"B",t2
print(t3)
```

All that is required to nest the existing tuples in a new tuple is to list the variables that reference the two existing tuples in a comma-separated list of items for creation of the new tuple. (*Note that the optional parentheses were omitted in [Listing 2](#).*)

[Figure 2](#) shows the output for the new tuple containing two nested tuples.

Figure 2 . Output from the code in Listing 2.

```
Create/print nested tuple  
( 'A', (1, 2), 'B', ('a', 'b'))
```

Note that the two nested tuples retain their identity as tuples, as indicated by the fact that the parentheses surrounding the items in the two nested tuples are preserved in the new tuple. This is also indicated by the [visualization](#) in [Figure 5](#).

Get the length of the new tuple

The code in [Listing 3](#) shows code that gets and displays the length of the new tuple that contains the two nested tuples.

Listing 3 . Get the length of the new tuple.

```
print("Length of nested tuple is:")  
print(len(t3))
```

The length is a measure of the number of items in the tuple, and is obtained using the method named **len** .

[Figure 3](#) shows the output produced by the code in [Listing 3](#), including the length of the new tuple containing the two nested tuples.

Figure 3 . Output from the code in Listing 3.

```
Length of nested tuple is:  
4
```

It is important to note that even though the tuple shown in [Figure 2](#) actually consists of six individual items (*ignoring parentheses*), each of the nested tuples is treated as a single item, giving a length of only four items for the tuple that contains the two nested tuples.

This would be true regardless of the length of the nested tuples.

You will learn in a future module that a double square-bracket indexing notation can be used to gain access to the individual items in tuples that are nested inside of other tuples.

Complete program listing

A complete listing of the program is shown in [Listing 4](#).

Listing 4 . Complete program listing.

Listing 4 . Complete program listing.

```
# Illustrates creating and displaying nested
tuples
#-----
-----

print("Create/print one tuple")
t1 = 1,2
print(t1)
print("Create/print another tuple")
t2 = "a","b"
print(t2)
print("Create/print nested tuple")
t3 = "A",t1,"B",t2
print(t3)
print("Length of nested tuple is:")
print(len(t3))
```

[Figure 4](#) shows a consolidated view of the output produced by this program.

Figure 4 . Output from the code in Listing 4.

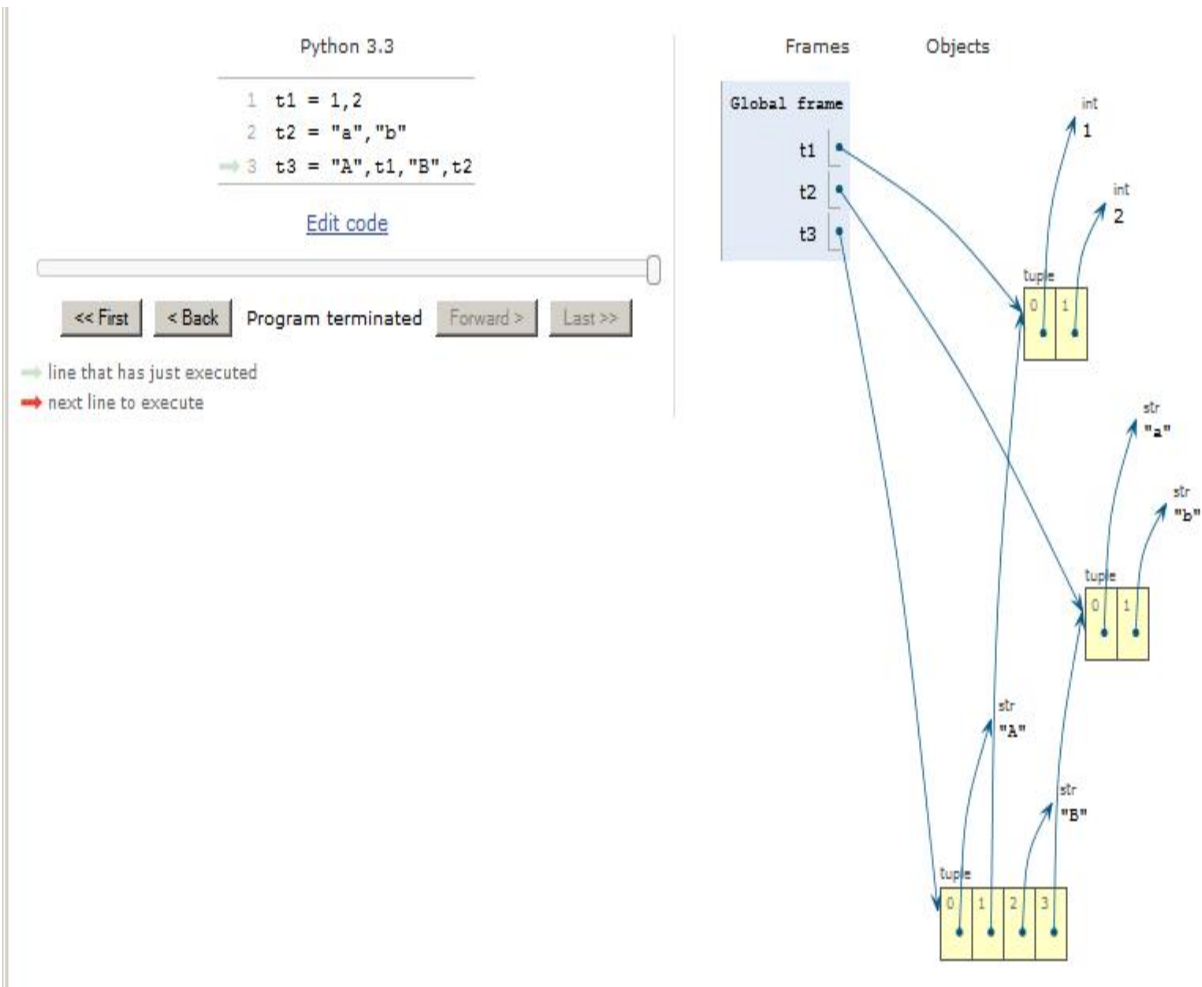
Figure 4 . Output from the code in Listing 4.

```
Create/print one tuple
(1, 2)
Create/print another tuple
('a', 'b')
Create/print nested tuple
('A', (1, 2), 'B', ('a', 'b'))
Length of nested tuple is:
4
```

Visualization of the tuples

A [visualization](#) of the tuples from this program is shown in [Figure 5](#).

Figure 5. Visualization of the tuples.



Run the program

I encourage you to copy the code from [Listing 4](#). Execute the code and confirm that you get the same results as those shown in [Figure 4](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1110-Nested Tuples
- File: Itse1359-1110.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1110r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1110-Nested Tuples.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#)
- [Figure index](#)
- [Listing index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1110-Nested Tuples* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? The program code in [Listing 1](#) produces the output shown in [Figure 1](#).

Listing 1 . Question 1 program code.

```
t1 = 1,2  
print(t1)  
  
t2 = "a","b"  
print(t2)  
  
t3 = "A",t1,"B",t2  
print(t3)
```

Figure 1 . Question 1 possible output.

Figure 1 . Question 1 possible output.

```
(1, 2)
('a', 'b')
('A', (1, 2), 'B', ('a', 'b'))
```

Go to [answer 1](#)

Question 2

True or False? The program code in [Listing 2](#) produces the output shown in [Figure 2](#).

Listing 2 . Question 2 program code.

```
t1 = 1,2
print(t1)

t2 = "a","b"
print(t2)

t3 = "A",t1,"B",t2
print(t3)
print(len(t3))
```

Figure 2 . Question 2 possible output.

```
(1, 2)
('a', 'b')
('A', (1, 2), 'B', ('a', 'b'))
6
```

Go to [answer 2](#)

Figure index

- [Figure 1](#). Question 1 possible output.
- [Figure 2](#). Question 2 possible output.
- [Figure 3](#). Question 2 actual output.

Listing index

- [Listing 1](#). Question 1 program code.
- [Listing 2](#). Question 2 program code.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 2

False. The actual output is shown in [Figure 3](#). The code in [Listing 2](#) produces a tuple containing two other nested tuples. A nested tuple counts

as only one element in the length of a tuple containing nested tuples.

Figure 3 . Question 2 actual output.

```
(1, 2)
('a', 'b')
('A', (1, 2), 'B', ('a', 'b'))
4
```

Go back to [Question 2](#)

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1110r-Review
- File: Itse1359-1110r.htm
- Published: 10/19/14

- Revised: 02/23/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1120-Empty and Single-Item Tuples

This module will teach you how to create empty tuples and tuples containing only one item.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
- [What is a tuple?](#)
- [Sample program](#)
 - [Empty, single-item, and nested tuples](#)
 - [An empty tuple](#)
 - [A tuple with only one element](#)
 - [Nested tuples](#)
 - [Doubly-nested tuples](#)
- [Complete program listing](#)
- [Visualization of the tuples](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Output from the code in Listing 1.
- [Figure 2](#). Output from the code in Listing 2.
- [Figure 3](#). Output from the code in Listing 3.
- [Figure 4](#). Consolidated output from the program in Listing 4.
- [Figure 5](#). Visualization of the tuples.

Listings

- [Listing 1](#). Beginning of the program.
- [Listing 2](#). A tuple with only one element.
- [Listing 3](#). Create and print nested tuples.
- [Listing 4](#). Create and print nested tuples.

Introduction

This module is part of a series of modules designed to teach you about tuples in Python.

Previous modules have illustrated

- How to create a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.

This module will teach you how to create empty tuples and tuples containing only one item.

What is a tuple ?

A tuple is like a list whose values cannot be modified. It is an ordered list of objects, and it can contain references to any type of object.

- Tuples are normally written as a sequence of items contained in *(optional)* matching parentheses.
- A tuple is an immutable sequence.
- Items in a tuple are accessed using a numeric index.
- Tuples can be nested.

Sample Program

Empty, single-item, and nested tuples

[Listing 4](#) shows a Python program that

- Creates an empty tuple.
- Creates a single-item tuple.
- Nests the two tuples along with some strings in a third tuple.
- Determines the length of each of the tuples.
- Displays all of the above

A consolidated output from the program is shown in [Figure 4](#).

A [visualization](#) of the tuples in the program is provided in [Figure 5](#).

An empty tuple

I will explain this program in fragments. The beginning of the program is shown in [Listing 1](#).

Listing 1 . Beginning of the program.

```
# Illustrates empty tuples and tuples with  
only one element  
#-----  
-----  
  
print("Create and print empty tuple")  
t1 = ()  
print(t1)  
print("Length of empty tuple is")  
print(len(t1))
```

As you might have guessed from the name, an empty tuple is just a pair of empty parentheses as shown by the second statement in [Listing 1](#). *(The parentheses aren't optional for an empty tuple.)*

[Figure 1](#) shows the output produced by the code in [Listing 1](#). The empty tuple is displayed simply as a pair of empty parentheses, and the length of the empty tuple is shown to be zero.

Figure 1 . Output from the code in Listing 1.

Figure 1 . Output from the code in Listing 1.

```
Create and print empty tuple  
(  
Length of empty tuple is  
0
```

A tuple with only one element

There are probably no surprises regarding an empty tuple. However, there may be some surprises in the code fragment shown in [Listing 2](#). This fragment deals with a tuple containing only one element.

Listing 2 . A tuple with only one element.

```
print("Create and print one-element tuple")  
# Note the req trailing comma  
t2 = "a",  
print(t2)  
print("Length of one-element tuple is:")  
print(len(t2))
```

The syntax for creating a tuple with only one element is rather ugly, but is required to avoid ambiguity. In particular, it is necessary to follow the

single tuple item with a comma as shown in the third line of text in [Listing 2](#).

Had I written that line simply as follows without the extra comma,

```
t2 = "a"
```

the result would have been to create a new variable named **t2** whose contents would be the string "a". *(The parentheses are optional here but the comma is required with or without the parentheses.)*

This would not indicate a tuple at all. The extra comma is required to make a single-item tuple unique and to distinguish it from other possibilities.

[Figure 2](#) shows the output produced by the code in [Listing 2](#). The single-item tuple is shown in the third line of text in [Listing 2](#). As is always the case, the tuple is displayed in parentheses.

Figure 2 . Output from the code in Listing 2.

```
Create and print one-element tuple
('a',)
Length of one-element tuple is:
1
```

The length of the tuple as shown in [Figure 2](#) is one (1) item.

Nested tuples

Just to give you a little more practice in dealing with nested tuples, the code in [Listing 3](#) nests the two tuples created above into a new tuple and stores a reference to the new tuple in the variable named **t3** .

Listing 3 . Create and print nested tuples.

```
print("Create and print nested tuple")
t3 = "A", t1, "B", (t2, "Z"), "C"
print(t3)

print("Length of nested tuple is")
print(len(t3))
```

Doubly-nested tuples

However unlike previous sample programs, in this case, literal parentheses are used to cause the tuple named **t2** to be doubly nested.

In particular, as shown by the second statement in [Listing 3](#), the tuple named **t2** and the string **"Z"** are used to create a tuple, which in turn, is nested in the tuple assigned to the variable named **t3** . This is also shown in the [visualization](#) in [Figure 5](#) .

The double nesting is evidenced by the extra parentheses in the second line of text in the output shown in [Figure 3](#) .

Figure 3 . Output from the code in Listing 3.

```
Create and print nested tuple  
( 'A', (), 'B', (('a',), 'Z'), 'C' )  
Length of nested tuple is  
5
```

The length of the tuple is also shown in [Figure 3](#). Even though the tuple named **t3** contains two nested tuples (*one of which is doubly-nested*) , its overall length is only five (5) items.

One of the tuples nested inside of **t3** has a length of zero but it still counts as one item when the length of **t3** is determined.

Complete program listing

A complete listing of the program is shown in [Listing 4](#).

Listing 4 . Complete program listing.

Listing 4 . Complete program listing.

```
# Illustrates empty tuples and tuples with
only one element
#-----
-----

print("Create and print empty tuple")
t1 = ()
print(t1)
print("Length of empty tuple is")
print(len(t1))

print("Create and print one-element tuple")
# Note the req trailing comma
t2 = "a",
print(t2)
print("Length of one-element tuple is:")
print(len(t2))

print("Create and print nested tuple")
t3 = "A",t1,"B",(t2,"Z"),"C"
print(t3)

print("Length of nested tuple is")
print(len(t3))
```

A consolidated output from the program is shown in [Figure 4](#).

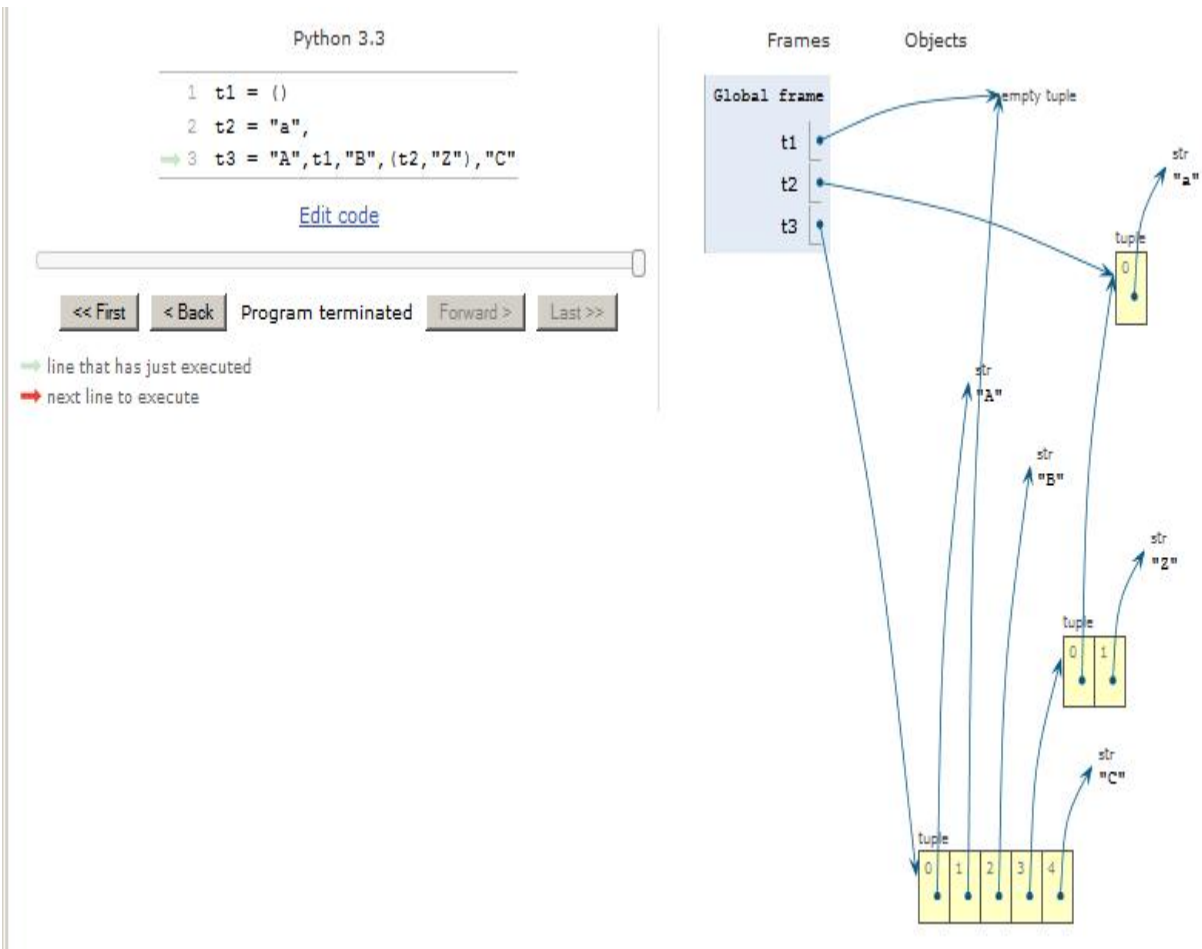
Figure 4 . Consolidated output from the program in Listing 4.

```
Create and print empty tuple
()
Length of empty tuple is
0
Create and print one-element tuple
('a',)
Length of one-element tuple is:
1
Create and print nested tuple
('A', (), 'B', (('a',), 'Z'), 'C')
Length of nested tuple is
5
```

Visualization of the tuples

A [visualization](#) of the tuples in the program is provided in [Figure 5](#).

Figure 5. Visualization of the tuples.



Run the program

I encourage you to copy the code from [Listing 4](#). Execute the code and confirm that you get the same results as those shown in [Figure 4](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1120-Empty and Single-Item Tuples
- File: Itse1359-1120.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1120r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1120-Empty and Single-Item Tuples.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1120-Empty and Single-Item Tuples* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? The code in [Figure 1](#) creates and prints an empty tuple producing the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
t1 =  
print(t1)  
  
print(len(t1))
```

Figure 2 . Question 1 possible output.

```
()  
0
```

Go to [answer 1](#)

Question 2

True or False? The code in [Figure 4](#) creates and prints an empty tuple producing the output shown in [Figure 5](#).

Figure 4 . Question 2 program code.

```
t1 = ()  
print(t1)  
  
print(len(t1))
```

Figure 5 . Question 2 possible output.

```
()  
0
```

Go to [answer 2](#)

Question 3

True or False? The code in [Figure 6](#) produces the output shown in [Figure 7](#).

Figure 6 . Question 3 program code.

```
t2 = ("a")  
print(t2)
```

Figure 7 . Question 3 possible output.

```
('a')
```

Go to [answer 3](#)

Question 4

True or False? The code in [Figure 9](#) produces the output shown in [Figure 10](#) .

Figure 9 . Question 4 program code.

```
t2 = ("a", )  
print(t2)
```

Figure 10 . Question 4 possible output.

```
('a', )
```

Go to [answer 4](#)

Question 5

True or False? The code in [Figure 11](#) produces the output shown in [Figure 12](#).

Figure 11 . Question 5 program code.

Figure 11 . Question 5 program code.

```
t1 = ()
print(t1)
print(len(t1))

t2 = ("a",)
print(t2)
print(len(t2))

t3 = "A", t1, "B", (t2, "Z"), "C"
print(t3)
print(t3[3][0])
print(len(t3))
```

Figure 12 . Question 5 possible output.

```
()
0
('a',)
1
('A', (), 'B', (('a',), 'Z'), 'C')
('a',)
5
```

Go to [answer 5](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.
- [Figure 4](#). Question 2 program code.
- [Figure 5](#). Question 2 possible output.
- [Figure 6](#). Question 3 program code.
- [Figure 7](#). Question 3 possible output.
- [Figure 8](#). Question 3 actual output.
- [Figure 9](#). Question 4 program code.
- [Figure 10](#). Question 4 possible output.
- [Figure 11](#). Question 5 program code.
- [Figure 12](#). Question 5 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 5

True. Note the indexing using double square bracket notation.

Go back to [Question 5](#)

Answer 4

True.

Go back to [Question 4](#)

Answer 3

False. The output is shown in [Figure 8](#). The code in [Figure 6](#) doesn't recognize the item in parentheses as a tuple. It is being treated as an ordinary variable. It needs a comma to be recognized as a tuple.

Figure 8 . Question 3 actual output.

a

Go back to [Question 3](#)

Answer 2

True.

Go back to [Question 2](#)

Answer 1

False. The actual output is shown in [Figure 3](#). While the parentheses are optional when creating and populating a non-empty tuple, they are not optional when creating an empty tuple.

Figure 3 . Question 1 actual output.

```
File "1359-1120r-01.py", line 3
  t1 =
      ^
SyntaxError: invalid syntax
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1120r-Review
- File: Itse1359-1120r.htm
- Published: 10/19/14
- Revised: 02/23/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1130-Unpacking Tuples

This module will teach you about unpacking tuples in Python.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Introduction](#)
- [What is a tuple?](#)
- [Sample program](#)
 - [Program to unpack a tuple](#)
 - [Tuples can be concatenated](#)
 - [Unusual syntax](#)
 - [Packing and unpacking a tuple](#)
 - [Unpack the tuple into the mutable list](#)
- [Complete program listing](#)
- [Visualization of the tuples in the program](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- [Figure 1](#). Output from the code in Listing 1.
- [Figure 2](#). Output from the code in Listing 2.
- [Figure 3](#). Output from the code in Listing 3.
- [Figure 4](#). Output from the code in Listing 4.
- [Figure 5](#). Visualization of the tuples in the program.
- [Figure 6](#). Output from the modified program.

Listings

- [Listing 1](#). Beginning of the program.
- [Listing 2](#). Unpack the tuple and print individual elements.
- [Listing 3](#). Unpack the tuple into the mutable list.
- [Listing 4](#). Complete program listing.

Introduction

This module is part of a series of modules designed to teach you about tuples.

Earlier modules have illustrated

- How to create (*pack*) a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.

- How to create empty tuples.
- How to create single-item tuples.

This module will teach you about unpacking tuples.

What is a tuple ?

A tuple is an immutable ordered list of objects. It can contain references to any type of object. See earlier modules in this series for a more detailed description.

Sample program

Program to unpack a tuple

[Listing 4](#) presents a Python program that:

- Creates (packs) two simple tuples.
- Creates (packs) a third tuple by concatenating the first two tuples.
- Displays the third tuple.
- Unpacks the third tuple, assigning each item of the tuple into a separate variable.
- Displays each of the variables.
- Creates and displays a mutable list object containing five strings.
- Unpacks the tuple created earlier assigning the four items from the tuple into the first four items in the list.
- Displays the list.

[Figure 4](#) shows the output produced by the program in [Listing 4](#).

[Figure 5](#) shows a visualization of the tuples in the program after the first five statements in the code block have been executed. This is a case where you need to step through the program and observe changes in the diagram on the right to appreciate the behavior of the program.

I will explain this program in fragments. [Listing 1](#) shows the beginning of the program.

Listing 1 . Beginning of the program.

```
# Illustrates unpacking a tuple
#-----

# Create a pair of tuples
t1 = 1,2
t2 = "A", "B"

# Concatenate and print them
t3 = t1 + t2
print(t3)
```

Tuples can be concatenated

As shown in [Listing 1](#), tuples support the concatenation (+) operator. You can concatenate two or more tuples to produce a new tuple. This program creates two simple tuples, and then concatenates them to create a third tuple.

[Figure 1](#) shows the output produced by the code in [Listing 1](#). By now, the creation and display of simple tuples should be very familiar to you based on earlier modules. Therefore, I won't discuss this part of the program further.

Figure 1 . Output from the code in Listing 1.

Figure 1 . Output from the code in Listing 1.

```
(1, 2, 'A', 'B')
```

Unusual syntax

The code in [Listing 2](#) is not quite so straightforward. In fact, it looks rather strange if you come from a conventional C, C++, or Java programming background.

Listing 2 . Unpack the tuple and print individual elements.

```
# Unpack the tuple and print individual
elements
w,x,y,z = t3
print(w)
print(x)
print(y)
print(z)
```

[Figure 2](#) shows the output produced by the code in [Listing 2](#).

Figure 2 . Output from the code in Listing 2.

```
1  
2  
A  
B
```

If you compare this output with the original tuple in [Listing 1](#), or with the previous output in [Figure 1](#), you will see that each of the individual items in the tuple (*in left-to-right order*) were assigned respectively to the variables named **w** , **x** , **y** , and **z** .

Thus, the lines of output produced by printing these four variables in [Figure 2](#) match the items in the original tuple that was created in [Listing 1](#) and displayed in [Figure 1](#) .

Packing and unpacking a tuple

Version 3 of [The Python Tutorial -- 5.3. Tuples and Sequences](#) refers to the first, second, and third statements in [Listing 1](#) as tuple packing using the following example:

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible

An early version of [The Python Tutorial -- 5.3 Tuples and Sequences](#) refers to the first statement in [Listing 2](#) as *tuple unpacking* .

The current version of [The Python Tutorial -- 5.3. Tuples and Sequences](#) (September 2014) refers to that operation more generally as *sequence unpacking* .

That tutorial continues by telling us

"... sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence."

If you try to run a program that doesn't meet these criteria, you will get an error similar to the following:

ValueError: too many values to unpack (expected 3)

Unpack the tuple into the mutable list

Just to make things a little more interesting, I decided to combine the use of a tuple (*an immutable list*) and a regular mutable list in this program.

[Listing 3](#) contains the code to create a mutable list populated with five string characters.

Listing 3 . Unpack the tuple into the mutable list.

Listing 3 . Unpack the tuple into the mutable list.

```
# Create and print a list
L1 = ["a", "b", "c", "d", "e"]

# Unpack tuple into the list and print it
print(L1)
L1[0], L1[1], L1[2], L1[3] = t3
print(L1)
```

Then the list is displayed, as shown by the first line of text in [Figure 3](#). The first line of output in [Figure 3](#) shows the contents of the list just after it is created and populated.

Figure 3 . Output from the code in Listing 3.

```
['a', 'b', 'c', 'd', 'e']
[1, 2, 'A', 'B', 'e']
```

Then the code in [Listing 3](#) unpacks the four-element tuple referred to by **t3** into the first four elements in the list. (*Remember, a list is mutable, so the values of its items can be changed.*)

Then the contents of the list are displayed again producing the second line of text in [Figure 3](#). As you can see from [Figure 3](#), the first four items in the

list were replaced by the four items from the tuple. The fifth item in the list was not modified.

Complete program listing

A complete listing of the program discussed above is provided in [Listing 4](#).

Listing 4 . Complete program listing.

Listing 4 . Complete program listing.

```
# Illustrates unpacking a tuple
#-----

# Create a pair of tuples
t1 = 1,2
t2 = "A","B"

# Concatenate and print them
t3 = t1 + t2
print(t3)

# Unpack the tuple and print individual
elements
w,x,y,z = t3
print(w)
print(x)
print(y)
print(z)

# Create and print a list
L1 = ["a","b","c","d","e"]
# Unpack tuple into the list and print it
print(L1)
L1[0],L1[1],L1[2],L1[3] = t3
print(L1)
```

[Figure 4](#) shows the output from the code in [Listing 4](#).

Figure 4 . Output from the code in Listing 4.

```
(1, 2, 'A', 'B')  
1  
2  
A  
B  
['a', 'b', 'c', 'd', 'e']  
[1, 2, 'A', 'B', 'e']
```

Visualization of the tuples in the program

[Figure 5](#) shows a visualization of the tuples in the program after the first five statements in the code block have been executed. This is a case where you need to step through the program and observe changes in the diagram on the right to appreciate the behavior of the program.

Figure 5. Visualization of the tuples in the program.

Python 3.3

```
1 t1 = 1,2
2 t2 = "A","B"
3 t3 = t1 + t2
4 w,x,y,z = t3
5 → L1 = ["a","b","c","d","e"]
6 → L1[0],L1[1],L1[2],L1[3] = t3
```

[Edit code](#)

<< First

< Back

Step 6 of 6

Forward >

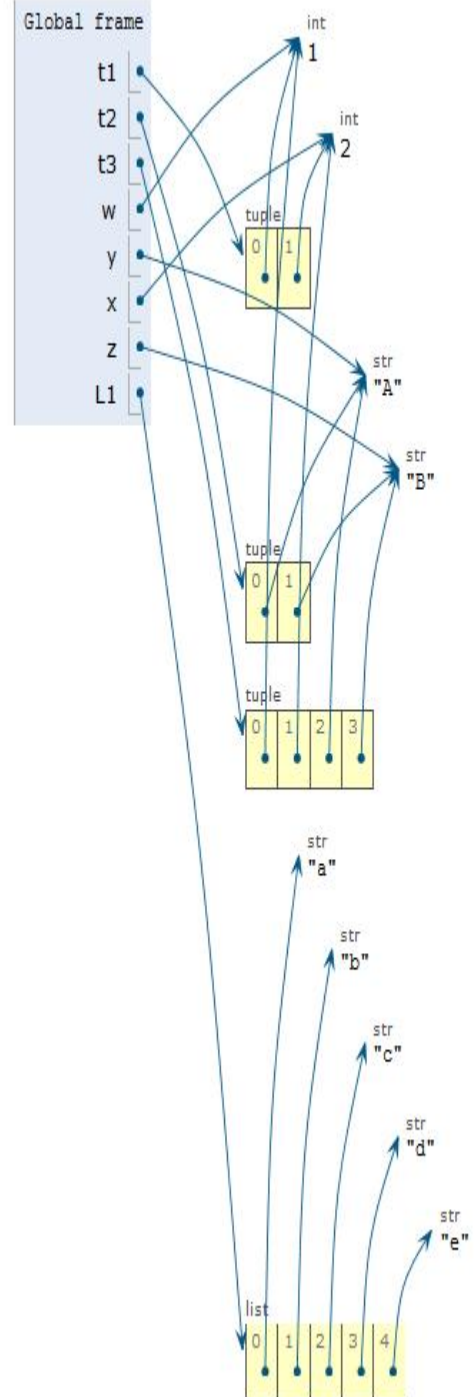
Last >>

→ line that has just executed

→ next line to execute

Frames

Objects



Run the program

I encourage you to copy the code from [Listing 4](#). Execute the code and confirm that you get the same results as those shown in [Figure 4](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

For example, you might want to try the following experiment. There is a line in the program shown in [Listing 4](#) that reads as follows:

```
L1 = ["a","b","c","d","e"]
```

Modify the program, changing this line so that it reads as follows:

```
L1 = ("a","b","c","d","e")
```

or

```
L1 = "a","b","c","d","e"
```

By now you should recognize that either of these modifications will change L1 from an ordinary mutable list to a tuple (*an immutable list*) .

Now execute the modified program. Your output should look something like that shown in [Figure 6](#).

Figure 6 . Output from the modified program.

Figure 6 . Output from the modified program.

```
(1, 2, 'A', 'B')
1
2
A
B
('a', 'b', 'c', 'd', 'e')
Traceback (most recent call last):
  File "1359-1130-07.py", line 24, in <module>
    L1[0],L1[1],L1[2],L1[3] = t3
TypeError: 'tuple' object does not support
item assignment
```

Everything should work well until the attempt is made to unpack the tuple named **t3** and to assign the items from that tuple into the individual items of the new tuple named **L1** .

The items in a tuple are immutable, meaning that they cannot be changed. Therefore, the program crashes at this point with the error shown in [Figure 6](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1130-Unpacking Tuples
- File: Itse1359-1130.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1130r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1130- Unpacking Tuples.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1230-The if Statement* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
dogs = ["Affenpinscher", "Afgan Hound", "Akita"]

if len(dogs) != 2:
    print(dogs[0])
    print(dogs[1])
    print(dogs[2])
print("Length of dogs list = " +
      str(len(dogs)))

if not(len(dogs) == 3):
    print(dogs[0])
    print(dogs[1])
    print(dogs[2])
print(dogs)
```

Figure 2 . Question 1 possible output.

```
Affenpinscher  
Afgan Hound  
Akita  
Length of dogs list = 3  
['Affenpinscher', 'Afgan Hound', 'Akita']
```

Go to [answer 1](#)

Question 2

True or False? The code in [Figure 3](#) produces the output shown in [Figure 4](#).

Figure 3 . Question 2 program code.

Figure 3 . Question 2 program code.

```
weather = ["sunshine","rain"]  
  
weatherToday = weather[1]  
  
if weatherToday == "rain":  
    print("It's raining, visit the museum.")  
else:  
    print("Sunshine, go to the beach.");  
  
print("Vacation is over, go home.")
```

Figure 4 . Question 2 possible output.

```
Sunshine, go to the beach.  
Vacation is over, go home.
```

Go to [answer 2](#)

Question 3

True or False? The code in [Figure 6](#) produces the output shown in [Figure 7](#).

Figure 6 . Question 3 program code.

```
weather = ["sunshine","rain","snow"]  
  
weatherToday = weather[2]  
  
if weatherToday == "rain":  
    print("It's raining, visit the museum.")  
elif weatherToday == "sunshine":  
    print("Sunshine, go to the beach.")  
else:  
    print("It's snowing, go skiing.")  
  
print("Vacation is over, go home.")
```

Figure 7 . Question 3 possible output.

```
It's snowing, go skiing.  
Vacation is over, go home.
```

Go to [answer 3](#)

Figure index

- [Figure 1](#). Question 1 program code.

- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 2 program code.
- [Figure 4](#). Question 2 possible output.
- [Figure 5](#). Question 2 actual output.
- [Figure 6](#). Question 3 program code.
- [Figure 7](#). Question 3 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 3

True.

Go back to [Question 3](#)

Answer 2

False. The actual output is shown in [Figure 5](#).

Figure 5 . Question 2 actual output.

Figure 5 . Question 2 actual output.

It's raining, visit the museum.
Vacation is over, go home.

Go back to [Question 2](#)

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1230r-Review
- File: Itse1359-1230r.htm
- Published: 10/21/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1210-The while Loop

This module introduces control flow in general and explains the use of the while loop in Python.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Repetition](#)
 - [Modularity](#)
 - [Decision logic](#)
- [Discussion and sample code](#)
 - [The while loop](#)
 - [Visualization of the program](#)
 - [Create, populate, and display a list](#)
 - [Create and initialize a counter variable](#)
 - [Execute a while loop](#)
 - [Append values to the end of the list](#)
 - [Increment the counter variable](#)
 - [Print the current state of the list](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Earlier modules in this collection have included instruction on the following topics:

- numbers
- variables
- identifiers
- strings
- scripts
- lists
- tuples

This has prepared you to use Python as a very fancy desk calculator.

What you will learn

The next step in learning to program with Python is to learn how to apply repetition, modularity, and decision logic to your scripts. This is often referred to in computer jargon as *control flow* .

This module introduces *control flow* in general and explains the use of the **while** loop in Python as an example of *control flow* .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Output from the script in Listing 1.
- [Figure 2](#). Output from the script in Listing 2.
- [Figure 3](#). Visualization of the program.

Listings

- [Listing 1](#). Typical script from earlier modules.
- [Listing 2](#). Use a while loop to manipulate a list.

General background information

A typical script in the earlier modules looked something like the one shown in [Listing 1](#).

Listing 1 . Typical script from earlier modules.

```
# Illustrates extracting a  
# list element and extracting  
# elements from a nested list  
#
```

Listing 1 . Typical script from earlier modules.

```
#-----  
print("Create and print a list")  
listA = [100,200,300,400,500]  
print(listA)  
print("Original length is:")  
print(len(listA))  
print("Replace an element")  
listA[2] = [2,4,8,16,32,64]  
print("Print the modified list")  
print(listA)  
print("Modified length is:")  
print(len(listA))  
print("Extract and display each")  
print(" element in the list")  
print(listA[0])  
print(listA[1])  
print(listA[2])  
print(listA[3])  
print(listA[4])  
  
print("Extract and display each")  
print(" element in nested list")  
print(listA[2][0])  
print(listA[2][1])  
print(listA[2][2])  
print(listA[2][3])  
print(listA[2][4])  
print(listA[2][5])
```

The output produced by the script in [Listing 1](#) is shown in [Figure 1](#).

Figure 1 . Output from the script in Listing 1.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
Extract and display each
element in the list
100
200
[2, 4, 8, 16, 32, 64]
400
500
Extract and display each
element in nested list
2
4
8
16
32
64
```

While the code in [Listing 1](#) does some interesting and useful things, it is missing one major component of modern computer programming -- ***control flow*** .

Control flow is generally considered to include the following features plus some others that I will get into in future modules:

- repetition
- modularity
- decision logic

Repetition

Repetition is the ability to cause specified portions of the code to be executed repeatedly under tightly controlled conditions. Repetition almost always embeds decision logic in some form. This module will introduce you to one of the mechanisms for implementing repetition in Python -- the **while** loop. Other mechanisms will be discussed in future modules.

Modularity

Modularity is the ability to subdivide the code into separate reusable units often called *functions* and *methods* . The design of functions and methods will be discussed in future modules.

Decision logic

Decision logic is the ability to make decisions on the basis of the program state and to determine which code will be executed as well as when and how it will be executed. As mentioned above, repetition almost always embeds decision logic in some form. In addition, Python supports a stand-alone decision capability -- the **if** statement, which will be discussed in a future module.

Discussion and sample code

The while loop

The **while** loop is the most fundamental and the most general of all the repetition mechanisms in most programming languages. Most programming languages provide other more specialized looping mechanisms for convenience, and Python is no exception to that rule. However, the fundamental behavior of most and perhaps all of those more specialized looping mechanisms can be replicated with a properly-configured **while** loop.

The code in [Listing 2](#) illustrates the use of a **while** loop. It also illustrates one of the features of **lists** -- the **append** method. Finally, it illustrates a decision structure that is embedded in the **while** loop.

Listing 2 . Use a while loop to manipulate a list.

```
# Illustrates the use of a while loop to
manipulate a list

aList = ["a", "b", "c"]
print(aList)

count = 0
while count <= 5:
    print(count)
    aList.append(count)
    count = count + 1
#   count += 1

print(aList)
```

The output produced by this program is shown in [Figure 2](#).

Figure 2 . Output from the script in Listing 2.

```
['a', 'b', 'c']  
0  
1  
2  
3  
4  
5  
['a', 'b', 'c', 0, 1, 2, 3, 4, 5]
```

Visualization of the program

[Figure 3](#) shows shows a [visualization](#) of the program after stepping through several iterations of the **while** loop. This static image doesn't do justice to the visualization. You really need to run it yourself and step through the program to appreciate it.

Figure 3. Visualization of the program.

Python 3.3

```

1 aList = ["a","b","c"]
2 print(aList)
3
4 count = 0
5 while count <= 5:
6     print(count)
7     aList.append(count)
8     count = count + 1
9     # count += 1
10
11 print(aList)

```

[Edit code](#)

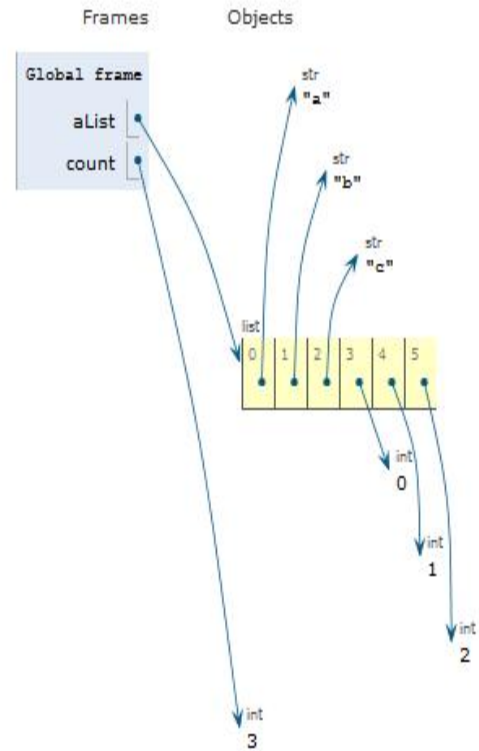
Step 16 of 29

Program output:

```

['a', 'b', 'c']
0
1
2

```



Create, populate, and display a list

The code in [Listing 2](#) begins by creating a list named `aList`. The list is initially populated with the strings "a", "b", and "c". Then that list is printed producing the first line of output text shown in [Figure 2](#).

Create and initialize a counter variable

Following that, a variable named `count` is created and initialized with the value 0. This variable will be used to control the number of iterations in the

while loop.

Execute a while loop

The next four lines of code in [Listing 2](#) constitute a **while** loop. The first line in the **while** loop contains a decision structure terminated by a colon.

Recall that the initial value of **count** is 0. The behavior of the first line in the **while** loop can be paraphrased as follows:

While the value of **count** is less than or equal (*note the relational operator that consists of a left angle bracket and an equal character*) to the literal value 5, execute all of the statements that follow the colon at the same indentation level.

Stated differently, for as long as the conditional clause (*count less than or equal to 5*) continues to be true, execute all of the statements that follow the colon at the same indentation level. When the conditional clause is no longer true, skip the indented statements and transfer control to whatever follows the indented statements.

In this case, there are three statements (*followed by a comment*) at the same indentation level following the colon. The first of the three statements prints the current value stored in the variable named **count** as shown by the second line of text in [Figure 2](#).

Append values to the end of the list

You learned in an earlier module that a list is a *mutable sequence* type. The term mutable means that the values stored in the list can be modified after

the list is created.

There are a dozen or more [operations](#) that can be performed on a list to modify its contents. One of those operations is **append(x)** . This operation appends the value of **x** to the end of the list. For the time being, I am going to refer to **append(x)** as a method belonging to the list object.

The second indented statement in [Listing 2](#) calls the **append** method on the list, appending the value currently stored in the variable named **count** to the end of the list. This will increase the length of the list by one.

Note that the **append** method is called on the list by joining the name of the list (**aList**) to the name of the **append** method using a period as a joining operator. The item that is to be appended to the list (*the value of **count***) is passed as a parameter to the **append** method.

Increment the counter variable

The last statement in the indented group of three statements increments or adds one to the value stored in the variable named **count** . This statement retrieves the value stored in the variable named **count** , adds one to that value, and stores the sum back in the variable named **count** replacing the value that was previously stored there.

The comment following that statement shows a shorthand way to accomplish the exact same thing.

Incrementing the **counter** variable is critical to the proper operation of the script. If the **counter** variable is not incremented within the body of the loop, it will remain less than 5 forever and the conditional clause at the top of the **while** loop (***count** less than or equal to 5*) will be true forever. This will cause the loop to continue looping forever. This is often referred to an *infinite loop* .

The code in [Listing 2](#) increments the counter variable once during each iteration of the loop. As you can see from the program output in [Figure 2](#), this causes it to count from 0 through 5 inclusive.

Each time the last indented statement in the body of the loop finishes executing, control goes back to the top of the loop where the value of **count** is tested against the literal value 5. *(You can see this happening by stepping through the visualization.)* When the count finally reaches 6,

- the conditional clause at the top of the loop will return **false** ,
- the three statements in the body of the loop will be skipped, and
- the **print** statement following the **while** loop structure will be executed.

Print the current state of the list

This causes the last statement in [Listing 2](#) to be executed. This statement prints the current state of the list. As you can see in [Figure 2](#), the length of the list has been increased from the original length of 3 elements to a new length of 9 elements. The list now contains the three original strings plus six new elements, which are the counter values from 0 through 5 inclusive.

Run the program

I encourage you to copy the code from [Listing 2](#). Execute the code and confirm that you get the same results as those shown in [Figure 2](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

I also encourage you to create and step through the [visualization](#) shown in [Figure 3](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1210-The While Loop
- File: Itse1359-1210.htm
- Published: 10/26/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1210r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1210-The While Loop.

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1210-The While Loop* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
# Illustrates the use of a while loop to
manipulate a list

aList = ["a", "b", "c"]
print(aList)

count = 6
while count <= 5:
    count = count + 1
    print(count)
    aList.append(count)

print(aList)
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

```
['a', 'b', 'c']  
1  
2  
3  
4  
5  
6  
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Go to [answer 1](#)

Question 2

True or False? The code in [Figure 4](#) produces the output shown in [Figure 5](#).

Figure 4 . Question 2 program code.

Figure 4 . Question 2 program code.

```
# Illustrates the use of a while loop to
manipulate a list

aList = ["a", "b", "c"]
print(aList)

count = 0
while count <= 5:
    count = count + 1
    print(count)
    aList.append(count)

print(aList)
```

Figure 5 . Question 2 possible output.

Figure 5 . Question 2 possible output.

```
['a', 'b', 'c']  
1  
2  
3  
4  
5  
6  
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Go to [answer 2](#)

Question 3

True or False? The code in [Figure 7](#) produces the output shown in [Figure 8](#).

Figure 7 . Question 3 program code.

Figure 7 . Question 3 program code.

```
# Illustrates the use of a while loop to  
manipulate a list  
  
aList = ["a", "b", "c"]  
print(aList)  
  
count = 0  
while count <= 5:  
    count = count + 1  
    print(count)  
    aList.append(count)  
  
print(aList)
```

Figure 8 . Question 3 possible output.

Figure 8 . Question 3 possible output.

```
['a', 'b', 'c']  
1  
2  
3  
4  
5  
6  
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Go to [answer 3](#)

Question 4

True or False? The two statements shown in [Figure 9](#) produce the same result.

Figure 9 . Question 9.

```
count = count + 1  
  
count += 1
```

Go to [answer 4](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.
- [Figure 4](#). Question 2 program code.
- [Figure 5](#). Question 2 possible output.
- [Figure 6](#). Question 2 actual output.
- [Figure 7](#). Question 3 program code.
- [Figure 8](#). Question 3 possible output.
- [Figure 9](#). Question 9.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 4

True.

Go back to [Question 4](#)

Answer 3

True.

Go back to [Question 3](#)

Answer 2

False. The output is shown in [Figure 6](#). The indentation in [Figure 4](#) is not correct for the output shown in [Figure 5](#).

Figure 6 . Question 2 actual output.

```
['a', 'b', 'c']  
6  
['a', 'b', 'c', 6]
```

Go back to [Question 2](#)

Answer 1

False. The output is shown in [Figure 3](#). The code in the body of the while loop is never executed because the conditional clause returns false the first time it is tested.

Figure 3 . Question 1 actual output.

```
['a', 'b', 'c']  
['a', 'b', 'c']
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1210r-Review
- File: Itse1359-1210r.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1220-Operators

This module presents and explains some of the many operators used in Python.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Operators](#)
 - [Arithmetic operators](#)
 - [Relational operators](#)
 - [Logical operators](#)
 - [Bit shift operators](#)
 - [Miscellaneous operators](#)
- [Discussion and sample code](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Early modules in this collection included instruction on the following topics:

- numbers
- variables
- identifiers
- strings
- scripts
- lists

This instruction has prepared you to use Python as a very fancy desk calculator.

A more recent module provided instruction on the concepts of *repetition* and *decision logic* using the **while** loop as an example.

What you will learn

Prior to encountering the conditional clause in the **while** loop, all of the code involving operators was fairly self explanatory:

- addition
- subtraction
- multiplication
- division
- assignment

However, when examining the **while** loop, we were confronted with an operator that was less self explanatory -- the *less than or equal* operator. This suggests that this is the point in this collection where we need to describe all, or at least most of the operators used in Python. That is the purpose of this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Arithmetic operators.
- [Figure 2](#). Relational operators.
- [Figure 3](#). Logical operators.
- [Figure 4](#). Bit shift operators.
- [Figure 5](#). Miscellaneous operators.

Listings

- [Listing 1](#). Logical operators "and", "or", and "not".

Operators

The material on operators contained in this module was mainly extracted from [10.3.1. Mapping Operators to Functions](#) in [The Python Standard Library](#).

Arithmetic operators

As the name implies, the arithmetic operators are used for performing arithmetic. However, some of the operators are "overloaded" meaning that the behavior of the operator depends on the types of the operands. For example, as you learned in an earlier module, the (+) operator is used both for arithmetic addition and string concatenation.

[Figure 1](#) shows all or at least most of the arithmetic operators.

Figure 1 . Arithmetic operators.

+	Addition	a + b	
+	Positive	+ a	(does
	essentially nothing)		
-	Subtraction	a - b	
-	Negation	- a	(change sign of
	operand)		
*	Multiplication	a * b	
**	Exponentiation	a ** b	
/	Division	a / b	(floating point
			division)
//	Division	a // b	(integer
			division)
%	Modulo	a % b	

With the exception of (//) and (%), the behavior of all of the operators in [Figure 1](#) should be self-explanatory. The behavior of those two operators was explained in the earlier module titled *Itse1359-1020-Numbers* .

Relational operators

These are the operators that are typically used in the conditional clause of decision logic. Basically the operators test to determine if the operator describes the relationship between the left and right operands. If so, the expression returns true. Otherwise, it returns false.

[Figure 2](#) shows all, or at least most of the relational operators.

Figure 2 . Relational operators.

<	a < b	(true if a is less than b)
<=	a <= b	(true if a is less than or equal to b)
==	a == b	(true if a is equal to b)
!=	a != b	(true if a is not equal to b)
>=	a >= b	(true if a is greater than or equal to b)
>	a > b	(true if a is greater than b)

Logical operators

The logical operators are shown in [Figure 3](#).

Figure 3 . Logical operators.

Figure 3 . Logical operators.

and	And	a and b	(true if both a and b are true)
or	Inclusive Or	a or b	(true if either a or b are true)
not	Negation	not a	(switch a from true to false or from false to true)
&	Bitwise And	a & b	
^	Bitwise Exclusive Or	a ^ b	
	Bitwise Inclusive Or	a b	
~	Bitwise Inversion	~ a	

The operators in [Figure 3](#) are often used to embellish the relational operators from [Figure 2](#) in decision logic.

The first operator (***and***) returns true if both operands are true.

The second operator (***or***) returns true if either operand is true.

The third operator (***not***) has only one operand and it is on the right side of the operator. It switches the operand from true to false or from false to true.

The script in [Listing 1](#) shows examples of the use of these three operators in conjunction with the equality relational operator.

Listing 1 . Logical operators "and", "or", and "not".

```
a=5
print(a==5 and a==5)
print(a==5 and a==4)
print(a==5 and not a==5)
print(a==5 and not a==4)

print(a==5 or a==5)
print(a==5 or a==4)
print(a==4 or a==4)

#This script produces the following output:
True
False
False
True
True
True
False
```

The last four operators in [Figure 3](#) are used to perform logical operations at the bit level. An explanation of these operators is beyond the scope of this module.

Bit shift operators

The operators in [Figure 4](#) are used to shift bits to the right or left. As with the bitwise logical operators, a discussion of the behavior of these operators is beyond the scope of this module.

Figure 4 . Bit shift operators.

Left Shift	<code>a << b</code>
Right Shift	<code>a >> b</code>

Miscellaneous operators

[Figure 5](#) shows a variety of miscellaneous operators. You are already familiar with the behavior of the concatenation operator. The behavior of the remaining operators will be explained when and if they are used in future modules.

Figure 5 . Miscellaneous operators.

Figure 5 . Miscellaneous operators.

Concatenation	<code>seq1 + seq2</code>
Containment Test	<code>obj in seq</code>
Identity	<code>a is b</code>
Identity	<code>a is not b</code>
Indexed Assignment	<code>obj[k] = v</code>
Indexed Deletion	<code>del obj[k]</code>
Indexing	<code>obj[k]</code>
Slice Assignment	<code>seq[i:j] = values</code>
Slice Deletion	<code>del seq[i:j]</code>
Slicing <code>seq[i:j]</code>	
String Formatting	<code>s % obj</code>
Truth Test	<code>obj truth(obj)</code>

Discussion and sample code

Other than the code shown in [Listing 1](#), I won't present code to illustrate the use of these operators in this module. Instead, I will use the operators in code in future modules and discuss them, as appropriate when they are used.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1220-Operators
- File: Itse1359-1220.htm

- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1220r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1220-Operators.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1220-Operators*.

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
count = 0
while count <= 5 and 3*count//2 <=5:
    count = count + 1
    print(count)

print(count)
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

1
2
3
4
4

Go to [answer 1](#)

Question 2

True or False? The code in [Figure 3](#) produces the output shown in [Figure 4](#).

Figure 3 . Question 2 program code.

```
count = 0
while count < 5 or 2*count+1 < 15:
    count = count + 1
    print(count)

print(count)
```

Figure 4 . Question 2 possible output.

1
2
3
4
5
6
7
7

Go to [answer 2](#)

Question 3

True or False? The code in [Figure 5](#) produces the output shown in [Figure 6](#).

Figure 5 . Question 3 program code.

Figure 5 . Question 3 program code.

```
count = 0
while not(count >= 5 and 2*count+1 >= 15):
    count = count + 1
    print(count)

print(count)
```

Figure 6 . Question 3 possible output.

```
1
2
3
4
5
6
7
8
8
```

Go to [answer 3](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 2 program code.
- [Figure 4](#). Question 2 possible output.
- [Figure 5](#). Question 3 program code.
- [Figure 6](#). Question 3 possible output.
- [Figure 7](#). Question 3 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 3

False. The actual output is shown in [Figure 7](#).

Figure 7 . Question 3 actual output.

Figure 7 . Question 3 actual output.

1
2
3
4
5
6
7
7

Go back to [Question 3](#)

Answer 2

True.

Go back to [Question 2](#)

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1220r-Review
- File: Itse1359-1220r.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1230-The if Statement

This module explains Python decision logic in general and the if statement in particular.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Discussion and sample code](#)
 - [The if statement](#)
 - [Visualization of the code in Listing 1](#)
 - [Importing modules](#)
 - [The if...else statement](#)
 - [Visualization of the code in Listing 2](#)
 - [Nested if statements](#)
 - [Visualization of the code in Listing 3](#)
- [Run the programs](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You were introduced to decision logic in the earlier module titled *Itse 1359-1210-The While Loop* . In that module I told you that Python supports a stand-alone decision capability, the **if** statement, which would be discussed in a future module. This is that future module.

You learned about the relational and logical operators that are used in decision logic in the module titled *Itse1359-1220-Operators* .

What you will learn

In this module, you will learn about

- the if statement
- the if...else statement
- nested if statements

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#) . Pseudo-code for an if statement.
- [Figure 2](#) . Output from the code in Listing 1.
- [Figure 3](#) . Visualization of the code in Listing 1.

- [Figure 4](#). Pseudo-code for a Python if...else statement.
- [Figure 5](#). Output from the code in Listing 2.
- [Figure 6](#). Output from the code in Listing 3.

Listings

- [Listing 1](#). Example of if statement usage.
- [Listing 2](#). Example of if...else statement usage.
- [Listing 3](#). Example of nested if statements.

Discussion and sample code

The if statement

The pseudo-code for a Python **if** statement is shown in [Figure 1](#).

Figure 1 . Pseudo-code for an if statement.

```
if expression is True:  
    Execute one or more statements at same  
    indentation level  
Execute next statement
```

The expression must be one that will evaluate to either True or False.

If the expression evaluates to True, the indented statement(s) that follow the colon will be executed in sequence.

If the expression evaluates to False, those indented statements will be skipped and the next statement following the **if** statement will be executed.

[Listing 1](#) shows an example of the use of an **if** statement.

Listing 1 . Example of if statement usage.

```
# Illustrates the if statement
#
#-----

dogs = ["Affenpinscher", "Afgan Hound", "Akita"]

if len(dogs) > 2:
    print(dogs[0])
    print(dogs[1])
    print(dogs[2])
print("Length of dogs list = " +
      str(len(dogs)))

if len(dogs) == 2:
    print(dogs[0])
    print(dogs[1])
    print(dogs[2])

print(dogs)
```

The output from the code in [Listing 1](#) is shown in [Figure 2](#).

Figure 2 . Output from the code in Listing 1.

```
Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
['Affenpinscher', 'Afgan Hound', 'Akita']
```

[Listing 1](#) begins by creating and populating a list containing the names of three breeds of dogs. Then it tests to see if the length of the list is greater than 2, which it is.

Because the test returns True, each element in the list is printed in sequence by the three indented **print** statements.

After that, the length of the list is printed by the non-indented **print** statement. This requires that the numeric length of the list be obtained and converted to a string for printing.

The built-in function named [len](#) is called here (*and also in the conditional clause of the **if** statement*) to get the length of the list.

The built-in function named [str](#) is called to convert the numeric length of the list into a string for concatenation with another string for printing.

Then the process is repeated except instead of testing the length of the list for *greater than 2* , it is tested for *equal to 2* . This test returns False, which causes the indented **print** statements in the body of the **if** statement to be skipped.

After that, the list is printed producing the last line of text in [Figure 2](#) .

Visualization of the code in Listing 1

[Figure 3](#) shows a [visualization](#) of the code in [Listing 1](#). I recommend that you create this visualization on your own and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of an **if** statement.

Figure 3. Visualization of the code in Listing 1.

Python 3.3

```
1 dogs = ["Affenpinscher","Afgan Hound","Akita"]
2
3 if len(dogs) > 2:
4     print(dogs[0])
5     print(dogs[1])
6     print(dogs[2])
7 print("Length of dogs list = " + str(len(dogs)))
8
9 if len(dogs) == 2:
10    print(dogs[0])
11    print(dogs[1])
12    print(dogs[2])
13
➔ 14 print(dogs)
```

[Edit code](#)

<< First

< Back

Program terminated

Forward >

Last >>

➔ line that has just executed

➔ next line to execute

Program output:

Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
['Affenpinscher', 'Afgan Hound', 'Akita']

Frames

Objects

Global frame

dogs

list

0

1

2

str

"Affenpinscher"

str

"Afgan Hound"

str

"Akita"

Importing modules

The next section will use a function named **randint** belonging to a module named **random** . Therefore, I need to say a few words about modules at this point. I will discuss modules in more detail in a future module.

You learned about creating scripts in the earlier module titled [Itse1359-1050-Introduction to Scripts](#) . If you save a script in a file, you can *import* it into another script and use it later. When used in this way, a file containing a script is called a *module* .

The previous section touched on the fact that Python has a number of built-in functions such as **len** and **str** . Similarly, the standard Python distribution includes [The Python Standard Library](#) that includes a large number of pre-written modules. One of those modules is named **random** . The purpose of the **random** module is to generate pseudo-random numbers. The **random** module provides many functions, one of which is named **randint(a,b)** This function will return a random integer with a value between **a** and **b** inclusive.

All that is necessary to use a built-in function such as **len** or **str** is to call out its name as shown in [Listing 1](#) . I interpret this to mean that the built-in functions occupy memory any time that the Python interpreter has been loaded into memory (*but I may be wrong on that*) .

However, before you can use a module or the functions contained in a module, you must first *import* the module into memory. There are a variety of options available when importing modules in Python. You will see an example of one way to import a module and to use its functions in the next section.

The if...else statement

The **if** statement shown in [Listing 1](#) basically tells the program to test for a condition. If the condition is True, do something special and then continue business as usual. If the condition is False, don't do anything special -- just continue business as usual.

Sometimes our decision processes are more complicated than that. Sometimes we need the program to test for a condition and if the condition is True, the program should do something special. However, if the condition is False, the program should do something different that is also special.

After one or the other of the special things is done, the program should continue with business as usual. That is the purpose of the **if...else** statement.

The pseudo-code for a Python **if...else** statement is shown in [Figure 4](#).

Figure 4 . Pseudo-code for a Python if...else statement.

```
if expression is True:
    Execute statements at same indentation
    level
else:
    Execute different statements at same
    indentation level
Execute next statement
```

As before, the expression must be one that will evaluate to either True or False.

If the expression evaluates to True, the indented statement(s) that follow the expression will be executed in sequence and the indented statement(s) that follow **else:** will be skipped.

If the expression evaluates to False, the indented statement(s) that follow the expression will be skipped and the indented statements that follow **else:** will be executed.

After that, the next statement following the **if** statement will be executed.

An example **if...else** statement is shown in [Listing 2](#).

Listing 2 . Example of if...else statement usage.

```
# Illustrates the if...else statement
#-----

import random

weather = ["sunshine", "rain"]
rain = False
sunshine = False

# Loop until rain and sunshine are both true
while (rain == False or sunshine == False):

    # Get today's weather
    weatherToday = weather[random.randint(0,1)]

    if weatherToday == "rain":
        rain = True
        print("It's raining, visit the museum.")
    else:
        sunshine = True
        print("Sunshine, go to the beach.");

print("Vacation is over, go home.")
```

The code in [Listing 2](#) simulates a short vacation. You want to visit the museum at least once and go to the beach at least once before you go home.

The code begins by importing the **random** module as described [earlier](#). Then it creates and populates three working variables that will be used later.

A **while** loop is used to assure that you visit both the museum and the beach at least once before going home. Note the use of the equality

operator (`==`) and the logical (`or`) operator in the conditional clause of the **while** loop.

The first statement inside the **while** loop uses the random number generator to get the weather for the day from the list named **weather** . The result will be either "sunshine" or "rain".

Following that, an **if...else** statement is used to decide whether to *visit the museum* or to *go to the beach* based on the weather. Note the use of the equality operator (`==`) in the conditional clause of the **if** statement. Also note that the corresponding working variable (*rain or sunshine*) is set to **True** to confirm the visit for the benefit of the conditional clause of the **while** loop.

When the museum and the beach have each been visited at least once, the **while** loop will terminate causing the "go home" print statement to be executed.

Because of the use of a random number generator to control the weather, this program will produce a different output each time you run it. [Figure 5](#) shows the output from one such run.

Figure 5 . Output from the code in Listing 2.

```
Sunshine, go to the beach.  
Sunshine, go to the beach.  
Sunshine, go to the beach.  
It's raining, visit the museum.  
Vacation is over, go home.
```


In this case, the random number generator produced the following sequence of numbers: 0,0,0,1 causing the beach to be visited three times before the visit to the museum.

Visualization of the code in Listing 2

I recommend that you create a [visualization](#) on your own for the code in [Listing 2](#) and step through the program one instruction at a time. Once again, as you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of an **if...else** statement.

Nested if statements

There are three keywords that can be used with nested **if** statements:

- **if**
- **else**
- **elif**

You already know about **if** and **else** . The keyword **elif** is short for "*else if*" . Nested **if** statements can contain zero or more **elif** parts and the **else** part is optional.

The **if...else** construct allows the program to choose between two options. The use of **elif** makes it possible to write code that can choose among three or more options.

The program shown in [Listing 3](#) extends the vacation analogy to three options:

- rain -- visit the museum
- sunshine -- go to the beach
- snow -- go skiing

Once again, you won't go home until you have at least one opportunity to do each of the three activities in the above list.

Listing 3 . Example of nested if statements.

Listing 3 . Example of nested if statements.

```
# Illustrates nested if statements
#-----

import random

weather = ["sunshine", "rain", "snow"]
rain = False
sunshine = False
snow = False

# Loop until rain and sunshine and snow are
all true
while (rain == False or sunshine == False or
snow == False):
    # Get today's weather
    weatherToday = weather[random.randint(0,2)]

    if weatherToday == "rain":
        rain = True
        print("It's raining, visit the museum.")
    elif weatherToday == "sunshine":
        sunshine = True
        print("Sunshine, go to the beach.")
    else:
        snow = True
        print("It's snowing, go skiing.")
print("Vacation is over, go home.")
```

The key thing to note in [Listing 3](#) is the **elif** after the **if** and before the **else** . This extends the choices from two as shown in [Listing 2](#) to three. The

number of choices could be extended to more than three by inserting more **elif** options.

As before, the output will be different each time you run the program. One such output is shown in [Figure 6](#).

Figure 6 . Output from the code in Listing 3.

```
It's snowing, go skiing.  
It's raining, visit the museum.  
It's snowing, go skiing.  
It's raining, visit the museum.  
It's raining, visit the museum.  
It's raining, visit the museum.  
It's snowing, go skiing.  
Sunshine, go to the beach.  
Vacation is over, go home.
```

You should be able to compare the output with the code in [Listing 3](#) and determine the sequence of eight random numbers produced by the random number generator in this case.

Visualization of the code in Listing 3

Once again I recommend that you create a [visualization](#) on your own for the code in [Listing 3](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom.

That should help you to better understand the behavior of nested **if** statements.

Run the programs

I encourage you to copy the code from [Listing 1](#), [Listing 2](#), and [Listing 3](#). Execute the code and confirm that you get the same results as those shown in [Figure 2](#), [Figure 5](#), and [Figure 6](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1230-The if Statement
- File: Itse1359-1230.htm
- Published: 10/21/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com

showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1230r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1230-The if Statement

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#)
- [Image index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1230-The if Statement* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Image 1](#) produces the output shown in [Image 2](#).

Image 1 . Question 1 program code.

```
dogs = ["Affenpinscher", "Afgan Hound", "Akita"]

if len(dogs) != 2:
    print(dogs[0])
    print(dogs[1])
    print(dogs[2])
print("Length of dogs list = " +
      str(len(dogs)))

if not(len(dogs) == 3):
    print(dogs[0])
    print(dogs[1])
    print(dogs[2])
print(dogs)
```


Image 2 . Question 1 possible output.

```
Affenpinscher  
Afgan Hound  
Akita  
Length of dogs list = 3  
['Affenpinscher', 'Afgan Hound', 'Akita']
```

Go to [answer 1](#)

Question 2

True or False? The code in [Image 3](#) produces the output shown in [Image 4](#).

Image 3 . Question 2 program code.

Image 3 . Question 2 program code.

```
weather = ["sunshine","rain"]  
  
weatherToday = weather[1]  
  
if weatherToday == "rain":  
    print("It's raining, visit the museum.")  
else:  
    print("Sunshine, go to the beach.");  
  
print("Vacation is over, go home.")
```

Image 4 . Question 2 possible output.

```
Sunshine, go to the beach.  
Vacation is over, go home.
```

Go to [answer 2](#)

Question 3

True or False? The code in [Image 6](#) produces the output shown in [Image 7](#).

Image 6 . Question 3 program code.

```
weather = ["sunshine","rain","snow"]  
  
weatherToday = weather[2]  
  
if weatherToday == "rain":  
    print("It's raining, visit the museum.")  
elif weatherToday == "sunshine":  
    print("Sunshine, go to the beach.")  
else:  
    print("It's snowing, go skiing.")  
  
print("Vacation is over, go home.")
```

Image 7 . Question 3 possible output.

```
It's snowing, go skiing.  
Vacation is over, go home.
```

Go to [answer 3](#)

Image index

- [Image 1](#). Question 1 program code.

- [Image 2](#). Question 1 possible output.
- [Image 3](#). Question 2 program code.
- [Image 4](#). Question 2 possible output.
- [Image 5](#). Question 2 actual output.
- [Image 6](#). Question 3 program code.
- [Image 7](#). Question 3 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 3

True.

Go back to [Question 3](#)

Answer 2

False. The actual output is shown in [Image 5](#).

Image 5 . Question 2 actual output.

Image 5 . Question 2 actual output.

It's raining, visit the museum.
Vacation is over, go home.

Go back to [Question 2](#)

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1230r-Review
- File: Itse1359-1230r.htm
- Published: 10/21/14
- Revised: 12/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1240-The for Loop

This module explains the Python for loop and the Python range function

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Iterate on a sequence](#)
 - [The range function](#)
- [Visualize the programs](#)
- [Run the programs](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

In the most recent modules you learned about the **while** loop, **operators** , and the **if** statement. While studying that material, you learned some other things as well such as the concepts of repetition, modularity, and decision logic.

What you will learn

You will learn about the **for** loop and the **range** function in this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Syntax of a for loop statement.
- [Figure 2](#). Output from the code in Listing 2.

Listings

- [Listing 1](#). Application of a for loop to a list and a string.
- [Listing 2](#). Using the range function to control a for loop.

General background information

You learned about the **while** loop in the earlier module titled [Itse1359-1210-The while Loop](#). Although the syntax of the **while** loop in Python is different from the syntax of the **while** loop in other programming languages such as C, C++, C#, and Java, the behavior is very similar.

You learned about the **if** statement in the earlier module titled [Itse1359-1230-The if Statement](#). Once again, although the syntax of the **if** statement in Python is different from the syntax of the **if** statement in the other programming languages listed above, the behavior is very similar.

That brings us to the **for** loop in Python. Neither the syntax nor the behavior of the Python **for** loop is similar to the syntax or the behavior of the primary **for** loop in those other languages.

However, some of those other languages have added a secondary **for** loop in recent years with behavior that is similar to the Python **for** loop. Also, the **range** function can be used with a Python **for** loop to produce behavior that is similar to the behavior of the primary **for** loops in those other languages.

Discussion and sample code

Iterate on a sequence

Python's **for** loop statement iterates over the items of any sequence (*such as a list or a string*) , in the order that they appear in the sequence. The syntax of a **for** loop is shown in [Figure 1](#).

Figure 1 . Syntax of a for loop statement.

```
for var in sequence:  
    statement(s)
```

The words "**for**" and "**in**" are keywords in the sequence. By that, I mean that they don't change from one script to the next. The name of the variable **var**, the name of the sequence, and the statements are provided by the programmer and are subject to change from one script to the next.

When control enters the loop, the first item in the sequence is assigned to **var**. Then the statements in the body of the loop are executed. Those statements may or may not refer to the contents of **var**, but very often will.

After that, the next item in the sequence is assigned to **var** and the statements in the body of the loop are executed. This process continues until every item in the sequence has been processed.

After all of the items in the sequence have been processed, control transfers to the next statement following the **for** loop statement. *(You will learn in a future module that it is also possible to have an **else** clause on a **for** loop.)*

[Listing 1](#) shows the application of a **for** loop to a list and then to a string extracted from the list.

Listing 1 . Application of a for loop to a list and a string.

Listing 1 . Application of a for loop to a list and a string.

```
# Illustrates the for loop
#-----

dogs = ["Affenpinscher","Afgan Hound","Akita"]

for breed in dogs:
    print(breed)
print("Length of dogs list = " +
      str(len(dogs)))

for letter in dogs[2]:
    print(letter)
print(dogs[2])
#-----
-----
#The output from this script is shown below.
#Do not copy the following text if you copy
the script.
Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
A
k
i
t
a
Akita
```

The output produced by the code in the top half of [Listing 1](#) is shown by the bottom half of [Listing 1](#).

The script begins by creating and populating a list with the names of three breeds of dogs.

The first **for** loop in [Listing 1](#) iterates through the list extracting and printing the name of each breed on a new line. When the first **for** loop in [Listing 1](#) terminates, the code gets and prints the length of the list.

The second **for** loop gets and iterates through the string at index value 2 in the list ("Akita") . It prints each letter from that string on a new line. When that **for** loop terminates, the code in [Listing 1](#) prints the contents of the string.

The range function

The primary **for** loops in many other programming languages deal strictly with numeric sequences. The built-in **range** function, when used with a Python **for** loop makes it possible to emulate that behavior.

The **range** function returns an immutable sequence containing a series of increasing integer values. There are two overloaded versions of the range function:

- `range(stop)`
- `range(start,stop[,step])`

The version with a single parameter returns a sequence containing the integers beginning with 0 and ending at one less than the value of *stop* . For example **range(6)** returns the following sequence:

```
[0,1,2,3,4,5]
```

The version of the **range** function with two and optionally three parameters returns a sequence that begins with *start* , ends just short of *stop* , and optionally increments by *step* . For example, **range(1,6,2)** returns the following sequence:

```
[1,3,5]
```

The sequence returned by range can be used to cause a for loop to iterate on the basis of a numeric index in much the same way that the primary **for** loop in C, C++, C#, and Java behaves. This is illustrated by the program in [Listing 2](#).

Listing 2 . Using the range function to control a for loop.

```
# Illustrates the for loop and the range
function
#-----
---

sum = 0
for cnt in range(5):
    sum += cnt
    print("cnt = " + str(cnt) + ", sum = " +
str(sum))
print("=====")
#-----
-----

breed = "Affenpinscher"
for index in range(1,len(breed),2):
    print("Letter at index " + str(index) + " is
" + breed[index])
print(breed)
```

The output from the code in [Listing 2](#) is shown in [Figure 2](#).

Figure 2 . Output from the code in Listing 2.

```
cnt = 0, sum = 0
cnt = 1, sum = 1
cnt = 2, sum = 3
cnt = 3, sum = 6
cnt = 4, sum = 10
=====
Letter at index 1 is f
Letter at index 3 is e
Letter at index 5 is p
Letter at index 7 is n
Letter at index 9 is c
Letter at index 11 is e
Affenpinscher
```

The code in the top half of [Listing 2](#) calls the **range** function to get an immutable sequence containing the integers from 0 through 4 inclusive. The **for** loop iterates through that sequence, extracting, summing, and printing the integer values and the sum of the values contained in the sequence.

Note that the body of this **for** loop contains two statements at the same indentation level.

The code in the bottom half of [Listing 2](#) calls the **range** function to get an immutable sequence containing the following integers:

```
[1,3,5,7,9,11]
```

The **for** loop iterates through that sequence and uses the integers contained in the sequence to extract and print corresponding letters from the string "Affenpinscher".

Visualize the programs

I recommend that you create [visualizations](#) for the code in [Listing 1](#) and [Listing 2](#) and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of **for** loops both with and without the use of the **range** function.

Run the programs

I also encourage you to copy the code from [Listing 1](#) and [Listing 2](#). Execute the code and confirm that you get the same results as those shown in [Listing 1](#) and [Figure 2](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1240-The for Loop
- File: Itse1359-1240.htm
- Published: 10/21/14
- Revised: 09/05/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1240r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1240-The for Loop.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1240-The for Loop* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
dogs = ["Affenpinscher", "Afgan Hound", "Akita"]

for breed in dogs:
    print(breed)
print("Length of dogs list = " +
      str(len(dogs)))

for letter in dogs[2]:
    print(letter)
print(dogs[2])
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

```
Akita
Afgan Hound
Affenpinscher
Length of dogs list = 3
A
k
i
t
a
Akita
```

Go to [answer 1](#)

Question 2

True or False? The code in [Figure 4](#) produces the output shown in [Figure 5](#).

Figure 4 . Question 2 program code.

Figure 4 . Question 2 program code.

```
breed = "Dachshund"
for index in range(2,len(breed),3):
    print("Letter at index " + str(index) + " is " + breed[index])
print(breed)

sum = 0
for cnt in range(5):
    sum += 2*cnt
    print("cnt = " + str(cnt) + ", sum = " + str(sum))
```

Figure 5 . Question 2 possible output.

```
Letter at index 2 is c
Letter at index 5 is h
Letter at index 8 is d
Dachshund
cnt = 0, sum = 0
cnt = 1, sum = 2
cnt = 2, sum = 6
cnt = 3, sum = 12
cnt = 4, sum = 20
```

Go to [answer 2](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.
- [Figure 4](#). Question 2 program code.
- [Figure 5](#). Question 2 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 2

True.

Go back to [Question 2](#)

Answer 1

False. The actual output is shown in [Figure 3](#).

Figure 3 . Question 1 actual output.

Figure 3 . Question 1 actual output.

```
Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
A
k
i
t
a
Akita
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1240r-Review
- File: Itse1359-1240r.htm
- Published: 10/21/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1250-Nested Loops

This module explains nested loops in Python.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
- [Visualizing the program code](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You have learned how to construct **while** loops, **for** loops, and **if** statements in recent modules on *control flow* in Python.

What you will learn

In this module, you will learn that loops can be nested and how to nest them.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Syntax for nested while loops.
- [Figure 2](#). Syntax for nested for loops
- [Figure 3](#). Output from the code in Listing 1.

Listings

- [Listing 1](#). An example of nested while and for loops.

General background information

Python allows you to nest **while** loops and **for** loops inside of **while** loops and **for** loops to a greater depth than you are likely to need.

[Figure 1](#) shows the general syntax for nested **while** loops.

Figure 1 . Syntax for nested while loops.

```
while expression:
    while expression:
        while expression:
            statement(s)
        statement(s)
    statement(s)
```

[Figure 2](#) shows the general syntax of nested **for** loops.

Figure 2 . Syntax for nested for loops

```
for var in sequence:
    for var in sequence:
        statements(s)
    statements(s)
```

The syntax can become more complicated than that shown in either [Figure 1](#) or [Figure 2](#). This is particularly true when you mix while loops and for loops in a nested construct.

Discussion and sample code

[Listing 1](#) shows a program with a **while** loop nested inside of a **for** loop which in turn is nested inside of another **while** loop.

Listing 1 . An example of nested while and for loops.

```
# Illustrates nested loops-----  
-----  
  
leftDigit = 0  
rightDigit = 0  
  
while leftDigit < 3:  
    for middleDigit in range(3):  
        rightDigit = 0  
        print("") #blank line  
  
        while rightDigit < 3:  
            print(str(leftDigit) + "-" +  
str(middleDigit) + "-" + str(rightDigit))  
            rightDigit += 1  
        #end inner while loop  
  
    #end for loop  
  
    leftDigit += 1  
#end outer while loop
```

[Figure 3](#) shows the output from the code in [Listing 1](#). Note that the blank lines were inserted to make the material in [Figure 3](#) easier to read.

Figure 3 . Output from the code in Listing 1.

0-0-0
0-0-1
0-0-2

0-1-0
0-1-1
0-1-2

0-2-0
0-2-1
0-2-2

1-0-0
1-0-1
1-0-2

1-1-0
1-1-1
1-1-2

1-2-0
1-2-1
1-2-2

2-0-0

Figure 3 . Output from the code in Listing 1.

2-0-1

2-0-2

2-1-0

2-1-1

2-1-2

2-2-0

2-2-1

2-2-2

This program simulates a scaled-down version of the odometer in your car. This odometer, however, only has three digits and each digit is limited to the values of 0, 1, and 2. In other words, Each digit of the odometer rolls over to 0 when its value exceeds 2.

The program begins by creating two working variables named **leftDigit** and **rightDigit** and initializing their values to zero. These are essentially counter variables for the outermost **while** loop and the innermost **while** loop in [Listing 1](#). They are also the left-most and right-most digits shown in the columns of digits in [Figure 3](#).

It isn't necessary to create a separate working variable for the middle digit. That variable, named **middleDigit** is created as part of the **for** loop statement in [Listing 1](#).

If you compare the code in [Listing 1](#) with the output in [Figure 3](#), you will see that the **for** loop goes through its full range of iterations (3) for each iteration of the outermost **while** loop.

You will also see that the innermost **while** loop goes through its full range of iterations (3) for every iteration of the **for** loop. As a result, the innermost **while** loop goes through nine iterations for every iteration of the outermost **while** loop.

When you think about the behavior of nested loops, think of an odometer. That may help you to remember how they behave.

Visualizing the program code

I recommend that you create a [visualization](#) for the code in [Listing 1](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of nested loops.

Run the program

I also encourage you to copy the code from [Listing 1](#). Execute the code and confirm that you get the same results as those shown in [Figure 3](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do. For example, rewrite the program using all **while** loops, all **for** loops, or some other combination of **while** loops and **for** loops.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1250-Nested Loops
- File: Itse1359-1250.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1250r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1250-Nested Loops

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1250-Nested Loops* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
leftDigit = 0
rightDigit = 0

while leftDigit < 3:

    for middleDigit in range(3,0,-1):
        rightDigit = 0
        print("") #blank line

        while rightDigit < 3:
            print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
            rightDigit += 1
        #end inner while loop

    #end for loop

    leftDigit += 1
#end outer while loop
```

Figure 2 . Question 1 possible output.

0-0-0

0-0-1

0-0-2

0-1-0

0-1-1

0-1-2

0-2-0

0-2-1

0-2-2

1-0-0

1-0-1

1-0-2

1-1-0

1-1-1

1-1-2

1-2-0

1-2-1

1-2-2

2-0-0

2-0-1

2-0-2

2-1-0

2-1-1

2-1-2

2-2-0

Figure 2 . Question 1 possible output.

2 - 2 - 1

2 - 2 - 2

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

False. The actual output is shown in [Figure 3](#).

Figure 3 . Question 1 actual output.

0-3-0

0-3-1

0-3-2

0-2-0

0-2-1

0-2-2

Figure 3 . Question 1 actual output.

0-1-0

0-1-1

0-1-2

1-3-0

1-3-1

1-3-2

1-2-0

1-2-1

1-2-2

1-1-0

1-1-1

1-1-2

2-3-0

2-3-1

2-3-2

2-2-0

2-2-1

2-2-2

2-1-0

2-1-1

2-1-2

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1250r-Review
- File: Itse1359-1250r-Review.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1260-Loop Modifiers

This module explains the Python loop modifiers: else, continue, break, and pass.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Discussion and sample code](#)
 - [The else clause](#)
 - [Visualize loops with else clauses](#)
 - [The continue statement](#)
 - [Visualize the behavior of the continue statement](#)
 - [The break statement](#)
 - [Visualize the behavior of the break statement](#)
 - [The pass statement](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You have learned how to construct **while** loops, **for** loops, and **if** statements in recent modules on *control flow* in Python. In addition, you recently learned how to create *nested loop structures* .

What you will learn

In this module, you will learn how to add an **else** clause to **while** loops and **for** loops. In addition, once you start using nested loops, you need to know how to use the loop modifiers: **continue** and **break** . Those loop modifiers will be explained in this module.

Finally, this module will make a brief mention of the **pass** statement, which does nothing. If you need to write code that does nothing, the **pass** statement is for you.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#) . Output from the code in Listing 1.
- [Figure 2](#) . Output from the code in Listing 2.
- [Figure 3](#) . Output from the code in Listing 3.

Listings

- [Listing 1](#). Loops with else clauses.
- [Listing 2](#). Nested loops with a continue statement.
- [Listing 3](#). Nested loops with a break statement.

Discussion and sample code

The else clause

You learned about the **else** clause on an **if** statement in an earlier module titled *Itse1359-1230-The if Statement* .

Loop statements can also have an **else** clause. The **else** clause on a **for** loop is executed when the loop terminates through exhaustion of the elements in the list. The **else** clause on a **while** loop is executed when the loop terminates because the condition becomes **False** . The **else** clause is not executed when **the loop is terminated by a [break](#)** statement.

I will illustrate the **else** clause using a program that is similar to the odometer program that you learned about in the earlier module titled [Itse1359-1250-Nested Loops](#) . The program is shown in [Listing 1](#) .

Listing 1 . Loops with else clauses.

```
# Illustrates nested loops with else clauses
#-----
-----

leftDigit = 0
```

Listing 1 . Loops with else clauses.

```
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
    leftDigit += 1

    #Begin for loop
    for middleDigit in range(1,3):
        rightDigit = 0

        #Begin innermost while loop
        while rightDigit < 3:
            rightDigit += 1
            print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
        else:
            print("In else clause on inner while
loop")
        #end inner while loop with else clause

    else:
        print("In else clause on for loop")
    #end for loop with else clause

else:
    print("In else clause on outer while loop")
#end outer while loop with else clause

print("Goodbye")
```

The output from the code in [Listing 1](#) is shown in [Figure 1](#).

Figure 1 . Output from the code in Listing 1.

```
1-1-1
1-1-2
1-1-3
In else clause on inner while loop
1-2-1
1-2-2
1-2-3
In else clause on inner while loop
In else clause on for loop
2-1-1
2-1-2
2-1-3
In else clause on inner while loop
2-2-1
2-2-2
2-2-3
In else clause on inner while loop
In else clause on for loop
In else clause on outer while loop
Goodbye
```

You should already be familiar with most of the code in [Listing 1](#). Some of the statements have been rearranged relative to the odometer code in the earlier module and some of the limits in the conditions have been reduced. However, that was done simply to shorten and to improve the clarity of the printed output and is of no technical consequence.

The only thing that is new in [Listing 1](#) is the addition of an **else** clause to each of the loops. Note that the indentation level of the word **else** in each case is the same as the indentation level for the corresponding words **while** and **for**. For example, the last **else** clause in [Listing 1](#) belongs to the

outermost **while** loop that begins with the word **while** near the top of [Listing 1](#).

In this program, each of the **else** clauses simply prints a message indicating that it is being executed. Obviously, more substantive code could be placed in the **else** clause in a more significant program.

The innermost **while** loop in this program is executed twice for each iteration of its enclosing **for** loop. Similarly, the **for** loop is executed twice for each iteration of its enclosing **while** loop. The outermost **while** loop is executed once, the **for** loop is executed twice, and the innermost **while** loop is executed **four** times.

The **else** clause for a loop is executed each time the loop terminates (*with the exception of [break](#) as mentioned earlier*) . As you can see in [Figure 1](#), the **else** clause on the innermost **while** loop was executed four times, the **else** clause on the **for** loop was executed two times, and the **else** clause on the outermost **while** loop was executed once.

Visualize loops with else clauses

I recommend that you create a [visualization](#) for the code in [Listing 1](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of loops with else clauses.

The continue statement

If you execute a **continue** statement inside a **while** loop or a **for** loop, that will cause the current iteration of the loop to terminate and the next iteration to begin.

(Be careful that the code that updates the condition in a while loop is not bypassed by a continue statement. If you do, you will end up with an infinite loop.)

The use of a **continue** statement is illustrated in [Listing 2](#), which shows an **if** statement (*with a **continue** statement in the body of the **if** statement*) inserted into the innermost **while** loop of the program from [Listing 1](#).

Listing 2 . Nested loops with a continue statement.

```
# Illustrates nested loops with else and
# continue
# -----
# -----

leftDigit = 0
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
    leftDigit += 1

    #Begin for loop
    for middleDigit in range(1,3):
        rightDigit = 0

        #Begin innermost while loop
        while rightDigit < 3:
```

Listing 2 . Nested loops with a continue statement.

```
        rightDigit += 1
        if rightDigit == 2:
            print("continue")
            continue
        print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
        else:
            print("In else clause on inner while
loop")
        #end inner while loop with else clause

    else:
        print("In else clause on for loop")
        #end for loop with else clause

else:
    print("In else clause on outer while loop")
    #end outer while loop with else clause

print("Goodbye")
```

[Figure 2](#) shows the output produced by the code in [Listing 2](#). You should compare this output with the output shown in [Figure 1](#).

Figure 2 . Output from the code in Listing 2.

Figure 2 . Output from the code in Listing 2.

```
1-1-1
continue
1-1-3
In else clause on inner while loop
1-2-1
continue
1-2-3
In else clause on inner while loop
In else clause on for loop
2-1-1
continue
2-1-3
In else clause on inner while loop
2-2-1
continue
2-2-3
In else clause on inner while loop
In else clause on for loop
In else clause on outer while loop
Goodbye
```

Once during each iteration of the **while** loop in [Listing 2](#), a test is made to determine if the condition variable, **rightDigit**, is equal to 2. If so, the word *continue* is printed and a **continue** statement is executed.

The execution of the **continue** statement causes the current iteration of the loop to terminate and the next iteration to begin. This, in turn causes the **print** statement following the **if** statement to be skipped and the word *continue* to be printed in its place. You can easily see where this happens by comparing the output in [Figure 2](#) with the output in [Figure 1](#).

Visualize the behavior of the continue statement

I also recommend that you create a [visualization](#) for the code in [Listing 2](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of the **continue** statement.

The break statement

The **break** statement is much more drastic than the **continue** statement. Whereas a **continue** statement causes the current iteration of the loop to terminate, a **break** statement causes the entire loop to terminate.

This is illustrated in [Listing 3](#), which shows a **break** statement inserted in place of the **continue** statement from [Listing 2](#). The **print** statement was also modified, causing it to print *break* in place of *continue*.

Listing 3 . Nested loops with a break statement.

```
# Illustrates nested loops with else and break
#-----
-----

leftDigit = 0
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
```

Listing 3 . Nested loops with a break statement.

```
leftDigit += 1

#Begin for loop
for middleDigit in range(1,3):
    rightDigit = 0

    #Begin innermost while loop
    while rightDigit < 3:
        rightDigit += 1
        if rightDigit == 2:
            print("break")
            break
        print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
    else:
        print("In else clause on inner while
loop")
    #end inner while loop with else clause

else:
    print("In else clause on for loop")
    #end for loop with else clause

else:
    print("In else clause on outer while loop")
    #end outer while loop with else clause

print("Goodbye")
```

The output from the code in [Listing 3](#) is shown in [Figure 3](#). You should compare this with the output shown in [Figure 2](#).

Figure 3 . Output from the code in Listing 3.

```
1-1-1
break
1-2-1
break
In else clause on for loop
2-1-1
break
2-2-1
break
In else clause on for loop
In else clause on outer while loop
Goodbye
```

The output for the **continue** statement in [Figure 2](#) shows that only one iteration of the innermost **while** loop was impacted by the **if** statement and the **continue** statement in its body.

The output shown in [Figure 3](#) shows that once the **break** statement was executed, no further iterations of the innermost **while** loop were executed. In other words, execution of the **break** statement terminated the loop in its entirety.

[Figure 3](#) also shows that termination caused by a **break** statement also prevented the execution of the code in the **else** clause for the innermost **while** loop as indicated [earlier](#)..

The innermost **while** loop is still executed twice for each iteration of its enclosing **for** loop. However, the **while** loop terminates part of the way through the second iteration and the printed output that would be produced by the second and third iterations (see [Listing 3](#)) is missing from the output shown in [Figure 3](#).

Visualize the behavior of the break statement

I also recommend that you create a [visualization](#) for the code in [Listing 3](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of the **break** statement.

The pass statement

I promised that this module will make brief mention of the **pass** statement, which does nothing. This is that brief mention. If you need to write code that does nothing, the **pass** statement will do that for you.

Run the program

I encourage you to copy the code from [Listing 1](#), [Listing 2](#), and [Listing 3](#). Execute the code and confirm that you get the same results as those shown in [Figure 1](#), [Figure 2](#), and [Figure 3](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

I also recommend that you create and experiment with [visualizations](#) for the code in [Listing 1](#), [Listing 2](#), and [Listing 3](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1260-Loop Modifiers
- File: Itse1359-1260.htm
- Published: 10/26/14

- Revised: 09/05/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1260r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1260-Loop Modifiers.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1260-Loop Modifiers* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
leftDigit = 0
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
    leftDigit += 1

    #Begin for loop
    for middleDigit in range(1,3):
        rightDigit = 0

        #Begin innermost while loop
        while rightDigit < 3:
            rightDigit += 1
            if rightDigit == 2:
                print("break")
                pass
            print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
        else:
            print("In else clause on inner while
loop")
```


Figure 1 . Question 1 program code.

```
#end inner while loop with else clause

else:
    print("In else clause on for loop")
#end for loop with else clause

else:
    print("In else clause on outer while loop")
#end outer while loop with else clause

print("Goodbye")
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

```
1-1-1
break
1-2-1
break
In else clause on for loop
2-1-1
break
2-2-1
break
In else clause on for loop
In else clause on outer while loop
Goodbye
```

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

False. The output is shown in [Figure 3](#). Although the program prints "break", the code doesn't actually contain a **break** statement. Instead it

contains a **pass** statement.

Figure 3 . Question 1 actual output.

```
1-1-1
break
1-1-2
1-1-3
In else clause on inner while loop
1-2-1
break
1-2-2
1-2-3
In else clause on inner while loop
In else clause on for loop
2-1-1
break
2-1-2
2-1-3
In else clause on inner while loop
2-2-1
break
2-2-2
2-2-3
In else clause on inner while loop
In else clause on for loop
In else clause on outer while loop
Goodbye
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1260r-Review
- File: Itse1359-1260r.htm
- Published: 10/26/14
- Revised: 02/26/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1270-Functions

This module explains the basics of defining and calling functions in Python.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
- [Visualizing a simple function](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

So far under the general topic of *control flow* , you have learned how to construct **while** loops, **for** loops, and **if** statements. In addition, you have learned

- how to create *nested loop structures* ,
- how to add an **else** clause to **while** loops and **for** loops, and
- how to use the loop modifiers: **continue** and **break** .

What you will learn

This module will depart significantly from the above. In this module, you will learn the basics of defining and calling functions in Python.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Syntax for a Python function definition.
- [Figure 2](#). Output produced by the code in Listing 1.
- [Figure 3](#). Visualizing the code from Listing 1.

Listings

- [Listing 1](#). A simple function named getRoot.

General background information

According to [tutorialspoint -- Python Functions](#)

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called user-defined functions.

Again according to [tutorialspoint](#), **the rules for defining** a basic function are as follows:

- Function blocks begin with the keyword `def` followed by the function name and parentheses `(())`.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or **docstring** .
- The code block within every function starts with a colon `(:)` and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

The [docstring](#) mentioned above is a string literal. We will ignore the *docstring* in this module. We will learn about tools in a future module that use *docstrings* to automatically produce online or printed documentation.

We will begin with these simple rules in this module. A future module titled [Itse1359-1280-Function Arguments](#) will get into more complicated material involving optional ways to define and use function arguments.

The general syntax for a Python function definition is shown in [Figure 1](#).

Figure 1 . Syntax for a Python function definition.

```
def name( parameters ):  
    """optional docstring"""  
    codeBlock  
    optional return [expression]
```

Discussion and sample code

A simple function named **getRoot** is defined and called in [Listing 1](#).

Listing 1 . A simple function named getRoot.

Listing 1 . A simple function named getRoot.

```
# Illustrates a simple function definition
#-----

def getRoot(number,root):
    """Returns the nth root of a number"""
    theRoot = number**(1/root)
    return theRoot
#End function definition

#Call the function
print("square root of 2")
print(getRoot(2,2))
print("cube root of 27")
print(getRoot(27,3))
print("eighth root of 256")
print(getRoot(256,8))
print("sixteenth root of 65536")
print(getRoot(65536,16))
```

[Figure 2](#) shows the output produced by the code in [Listing 1](#).

Figure 2 . Output produced by the code in Listing 1.

Figure 2 . Output produced by the code in Listing 1.

```
square root of 2
1.4142135623730951
cube root of 27
3.0
eighth root of 256
2.0
sixteenth root of 65536
2.0
```

[Listing 1](#) defines a function that returns the nth root of a number. The first parameter specifies the number for which the root is to be obtained. The second parameter specifies which root is to be obtained.

The function definition in [Listing 1](#) complies with each of the rules listed [above](#).

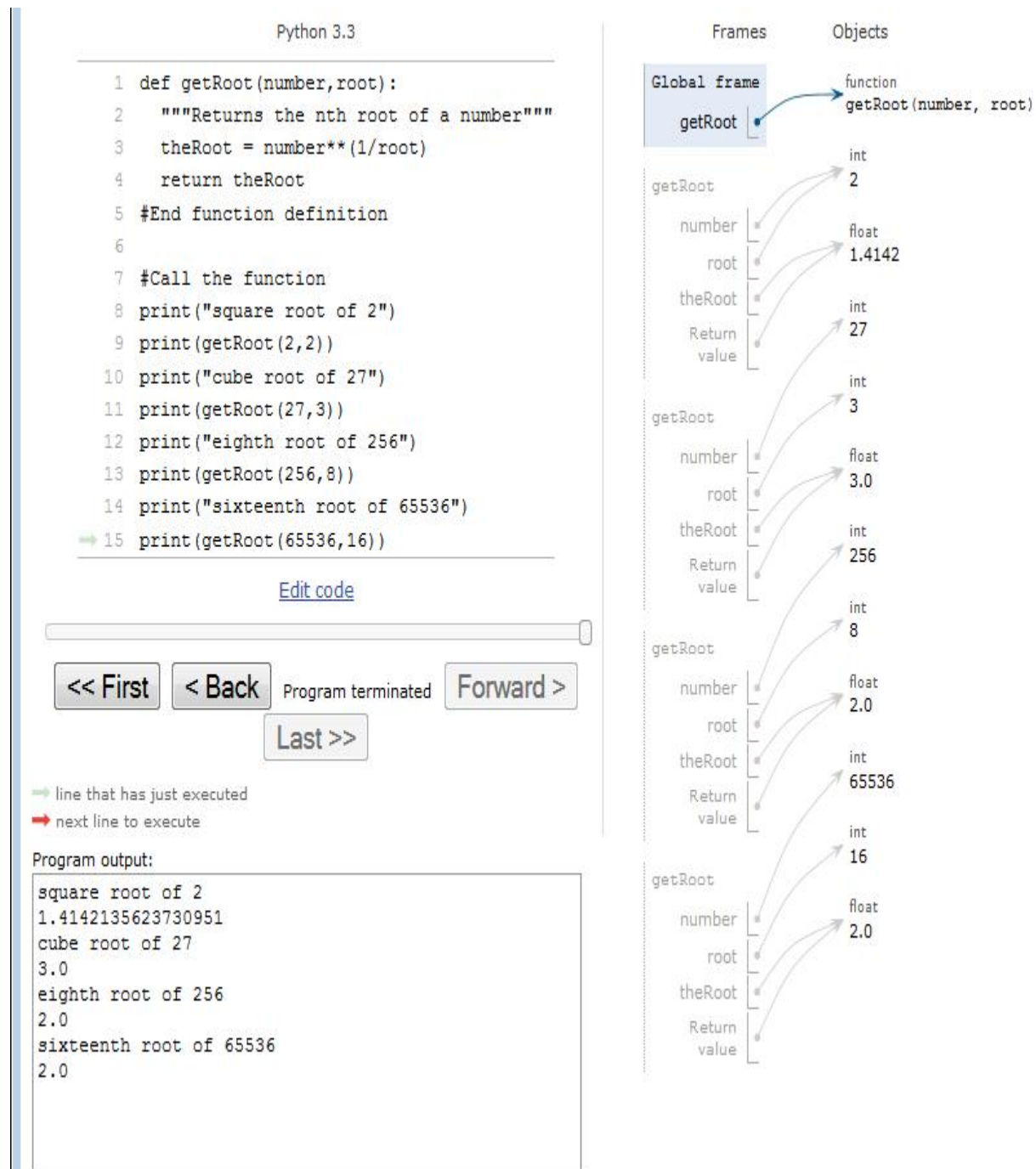
You should have no difficulty understanding the body of the function shown in [Listing 1](#) provided that you understand how to use exponentiation to compute a root of a number. Otherwise, you may need to dust off your old high school algebra book and do some remedial study of algebra.

If you are content to define and use simple functions with simple argument lists as in [Listing 1](#), that is about all you need to know about Python functions. However, if you want to tap into the more-powerful capabilities of functions, you will need to study the module titled [Itse1359-1280-Function Arguments](#) and possibly other modules as well.

Visualizing a simple function

[Figure 3](#) shows a [visualization](#) of the code from [Listing 1](#), which defines a simple function named **getRoot** and calls it several times.

Figure 3. Visualizing the code from Listing 1.



The following options were selected when performing this visualization:

1. show exited frames (Python)
2. render all objects on the heap
3. draw pointers as arrows

The first item in this list of options is what caused the four light gray occurrences of the calls to the **getRoot** function to be visible on the right in [Figure 3](#). *(If you were to change that option to "hide exited frames [default]", the diagram would show calls to the function while control is within the function but those calls would disappear when control leaves the function.)* The pointers associated with those occurrences point to the incoming parameter values and the return value for each call to the **getRoot** function.

I recommend that you create a [visualization](#) for the code in [Listing 1](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of Python programs containing functions that you define.

Run the program

I encourage you to copy the code from [Listing 1](#). Execute the code and confirm that you get the same results as those shown in [Figure 2](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1270-Functions
- File: Itse1359-1270.htm
- Published: 10/26/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1270r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1270-Functions.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1270-Functions* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
def getAnswer(number,power):  
    theAnswer = number**(power)  
    return theAnswer  
  
print(getAnswer(8,4))  
print(getAnswer(6,3))  
print(getAnswer(4,2))  
print(getAnswer(2,0))
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

4096
216
16
1

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1270r-Review
- File: Itse1359-1270r.htm
- Published: 10/26/14

- Revised: 02/27/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1280-Function Arguments

This module explains how to tap into some of the more-powerful capabilities of Python functions using required arguments, default arguments, keyword arguments, and variable-length arguments.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Passing by value or reference](#)
 - [The function definition](#)
 - [A non-technical explanation](#)
 - [Visualization of the code in Listing 1](#)
 - [A modified approach to the same objective](#)
 - [Required arguments](#)
 - [Default arguments](#)
 - [Keyword arguments](#)
 - [Variable-length arguments](#)
 - [Visualization of variable-length arguments](#)
- [Run the programs](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You learned the basics of defining and calling Python functions in the earlier module titled [Itse1359-1270-Functions](#).

What you will learn

In this module, you will learn how to tap into some of the more-powerful capabilities of functions involving optional ways to use arguments. This will include an explanation of

- required arguments,
- default arguments,
- keyword arguments, and
- variable-length arguments.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Output from the code in Listing 1.
- [Figure 2](#). Visualization of the code in Listing 1.
- [Figure 3](#). Output from the code in Listing 2.
- [Figure 4](#). Output produced by the code in Listing 3.
- [Figure 5](#). Output produced by the code in Listing 4.
- [Figure 6](#). Output produced by the code in Listing 5.
- [Figure 7](#). Visualization of variable-length arguments.

Listings

- [Listing 1](#). A list-modifier function.
- [Listing 2](#). Another list-modifier function.
- [Listing 3](#). Illustration of default arguments.
- [Listing 4](#). Illustration of keyword arguments.
- [Listing 5](#). Illustration of variable-length arguments.

General background information

The overall topic of function arguments in Python is broad and relatively complex. A complete treatment of the topic is beyond the scope of this collection of modules. Instead, I will provide some examples of what is possible. For a more complete treatment, I will refer you to [The Python Tutorial -- More on Defining Functions](#) and [tutorialspoint -- Python Functions](#).

One of the fundamental concepts in computer programming is whether function parameters are passed *by value* or *by reference*. If a parameter is passed by value, the function receives a copy of the thing referred to by the parameter. Modifying that thing in the function will not modify the original.

If a parameter is passed by reference, the incoming parameter can be used by code in the function to modify the original. Some programming languages support both types of parameter passing.

According to [tutorialspoint -- Python Functions](#), all parameters in Python are *passed by reference* . On the other hand, according to [The Python Tutorial -- Defining Functions](#),

"thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object)."

In this case, I come down on the side of *call by value* . I think it is more correct to say that parameters are passed by value and as a result, the function receives copies of references to objects. Even though I can use a copy of an object's reference to modify the object, I cannot use a copy of that reference to cause the original reference to point to a different object. I can modify the object to which the reference points, but I cannot modify the reference itself.

I will illustrate what I mean by this in conjunction with the program in [Listing 1](#).

Discussion and sample code

This module will examine the following kinds of arguments:

- Required arguments
- Default arguments
- Keyword arguments
- Variable-length arguments

Passing by value or reference

The program shown in [Listing 1](#) illustrates the significance of passing parameters by *value* or by *reference* when those parameters are references to objects. [Listing 1](#) also illustrates the use of two *required arguments* .

Listing 1 . A list-modifier function.

Listing 1 . A list-modifier function.

```
# Illustrates pass by value or reference
#-----
-----

def listModifier(listA,listB):
    """Illustrates pass by value or reference"""
    print("In listModifier")

    print("Use incoming parameter to append to
listA")
    listA.append(3.14159)
    print("New listA = " + str(listA))

    print("Assign a new list to listB")
    listB = ["A","new","list"]
    print("New listB = " + str(listB))

    return
#End function definition

#Call the function
print("Create two lists")
aList = ["ab","cd","ef"]
bList = ["The","old","list"]

print("aList = " + str(aList))
print("bList = " + str(bList))

print("Call listModifier")
listModifier(aList,bList)
print("Back from listModifier")
print("aList = " + str(aList))
print("bList = " + str(bList))
```

Listing 1 . A list-modifier function.

[Figure 1](#) shows the output produced by the code in [Listing 1](#).

Figure 1 . Output from the code in Listing 1.

```
Create two lists
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
Call listModifier
In listModifier
Use incoming parameter to append to listA
New listA = ['ab', 'cd', 'ef', 3.14159]
Assign a new list to listB
New listB = ['A', 'new', 'list']
Back from listModifier
aList = ['ab', 'cd', 'ef', 3.14159]
bList = ['The', 'old', 'list']
```

The function definition

The code in [Listing 1](#) defines a function named **listModifier** . This function receives two incoming parameters. Each parameter points to a different list. I contend that the incoming parameters are actually copies of the variables that were passed as parameters.

The code in the function uses one of the incoming parameters to access the list and to append a value of 3.14159 onto the end of the list. The second

line of text from the bottom of [Figure 1](#) confirms that this operation was successful.

Then the code in the function creates a new list and attempts to use the second incoming parameter to replace the original list pointed to by that parameter with the new list. The last line of text in [Figure 1](#) confirms that this was **not successful**. The copy of the incoming parameter points to the new list but this does not cause the original reference to point to the new list.

A non-technical explanation

This same situation occurs in the Java courses that I teach. I often attempt to explain it this way. Assume that I write my home address on a piece of paper and make a copy of that piece of paper. Then I lose the copy. If a burglar finds the copy, he has a pointer to my house. He could use that pointer in an attempt to break into my house. However, even if he were to scratch out the address on the copy and write a new address on the copy, that would not change the address written on the original. The original would still point to the same old address.

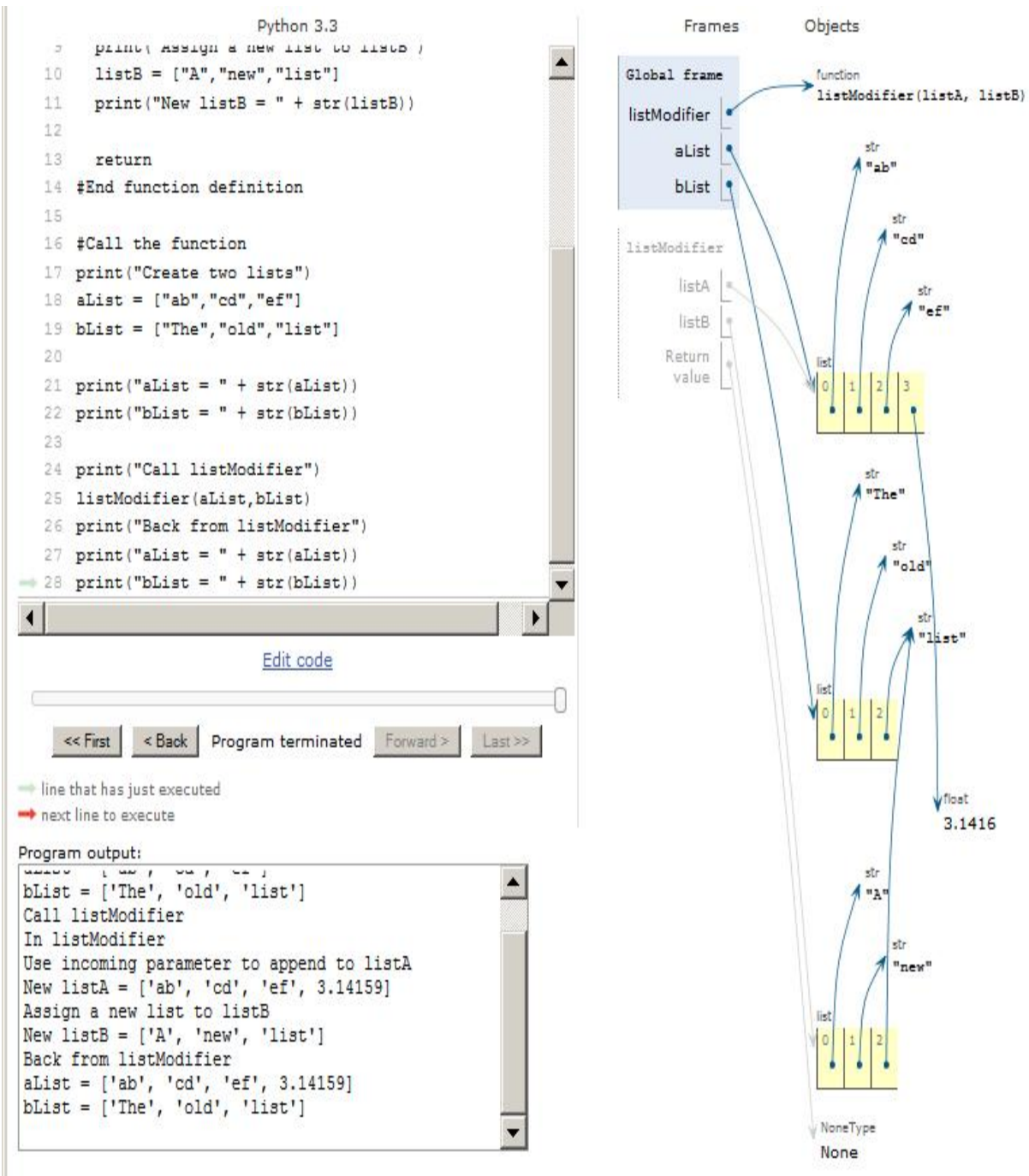
This is essentially what happens in [Listing 1](#) when the code creates a new list and assigns its value to the copy of the original reference. The copy points to the new list inside the function but the original reference outside the function continues to point to the original list.

In effect, the copies that are received as parameters by a function become local variables within the function. Those variables, and in this case the new list that is pointed to by one of those variables, all cease to exist when the function terminates unless returned by the function.

Visualization of the code in Listing 1

[Figure 2](#) shows a [visualization](#) of the code in [Listing 1](#) after all of the code has been executed.

Figure 2. Visualization of the code in Listing 1.



I recommend that you create a [visualization](#) for the code in [Listing 1](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the

diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of *"passing by value."*

A modified approach to the same objective

[Listing 2](#) shows a modification to the code from [Listing 1](#) that is designed to accomplish what [Listing 1](#) attempted to do but was unable to do.

Listing 2 . Another list-modifier function.

```
# Illustrates pass by value or reference
#-----
-----

def listModifier(listA,listB):
    """Illustrates pass by value or reference"""
    print("In listModifier")

    print("Use incoming parameter to append to
listA")
    listA.append(3.14159)
    print("New listA = " + str(listA))

    print("Assign a new list to listB")
    listB = ["A","new","list"]
    print("New listB = " + str(listB))

    return listB
#End function definition
```

Listing 2 . Another list-modifier function.

```
#Call the function
print("Create two lists")
aList = ["ab","cd","ef"]
bList = ["The","old","list"]

print("aList = " + str(aList))
print("bList = " + str(bList))

print("Call listModifier")
bList = listModifier(aList,bList)
print("Back from listModifier")
print("aList = " + str(aList))
print("bList = " + str(bList))
```

[Figure 3](#) shows the output from the code in [Listing 2](#).

Figure 3 . Output from the code in Listing 2.

Figure 3 . Output from the code in Listing 2.

```
Create two lists
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
Call listModifier
In listModifier
Use incoming parameter to append to listA
New listA = ['ab', 'cd', 'ef', 3.14159]
Assign a new list to listB
New listB = ['A', 'new', 'list']
Back from listModifier
aList = ['ab', 'cd', 'ef', 3.14159]
bList = ['A', 'new', 'list']
```

[Listing 2](#) has two major modifications relative to [Listing 1](#). First, the function returns a copy of the reference to the new list instead of returning nothing. Second, the return value is assigned to the variable named **bList**. This does cause the variable named **bList** to point to a different list object. This is confirmed by the last line of text in [Figure 3](#).

Conclusion -- function parameters in Python are passed by value, but the copies that are passed can often be used to access and modify the objects pointed to by the original reference variables of which they are copies.

In this program, the function's second argument is of no consequence. The function and the program would behave the same if the second argument were eliminated entirely. I left it in for illustration purposes only.

Hopefully you can study these two programs along with the program output and the above discussion to understand the basic behavior of Python function arguments.

Required arguments

Note that there is a subtle technical difference between arguments and parameters. However, I often tend to use the two terms interchangeably so you will simply need to grit your teeth and bear with me if you see me using one or the other term incorrectly.

When a function defines one or more arguments using the syntax shown in [Listing 1](#) and [Listing 2](#), the code that calls the function must pass one parameter for each argument and must pass them in the correct order. That is why we call them *required arguments*. There isn't a lot more that needs to be said about required arguments.

Default arguments

A function can define one or more default arguments in addition to required arguments as shown in [Listing 3](#). Default values are defined for the default arguments when the function is defined using the syntax shown in [Listing 3](#).

The calling program can ignore some or all of the default arguments when calling the function. By this I mean that the calling program can simply not pass parameters for some or all of the default arguments. However, when default arguments are ignored, they must be ignored from right to left in the argument list. In other words, the calling program cannot ignore a default argument in the middle of a group of default arguments.

When the calling program ignores default arguments, the defined default values for those arguments are used by the code in the body of the function.

The program shown in [Listing 3](#) illustrates a function with one required argument (**listA**) and three default arguments (**listB** , **listC** , and **listD**).

Listing 3 . Illustration of default arguments.

```
# Illustrates default arguments
#-----

def listModifier(listA,listB=["B"],listC=
["C"],listD=["D"]):
    """Illustrates default arguments"""
    print("In listModifier")

    listA.append(1.00001)
    print("listA = " + str(listA))
    listB.append(2.00002)
    print("listB = " + str(listB))
    listC.append(3.00003)
    print("listC = " + str(listC))
    listD.append(4.00004)
    print("listD = " + str(listD))

    return
#End function definition

aList = ["ab","cd","ef"]
bList = ["The","old","list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList,bList)
```

Listing 3 . Illustration of default arguments.

```
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))
```

[Figure 4](#) shows the output produced by the code in [Listing 3](#).

Figure 4 . Output produced by the code in Listing 3.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
listA = ['ab', 'cd', 'ef', 1.00001]
listB = ['The', 'old', 'list', 2.00002]
listC = ['C', 3.00003]
listD = ['D', 4.00004]
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house']
dList = ['is falling down']
```

The first thing that you should pay attention to is the syntax for defining default arguments in the function named **listModifier** . *(You will see later that this **same syntax is used for defining keyword arguments** .)*

The second thing you should pay attention to is the syntax used to call the function named **listModifier** about six lines up from the bottom of [Listing 3](#) . The function call passes only two parameters to the function: **aList** and **bList** .

The function treats the first parameter as satisfying the required argument named **listA** . The function treats the second parameter as satisfying the default argument named **ListB** , replacing the default value with the parameter that is actually received. Not receiving any more incoming parameters, the function processes the default values for **listC** and **listD** .

Hopefully this explanation along with the code in [Listing 3](#) and the output in [Figure 4](#) will tell you what you need to know about using default function arguments.

Keyword arguments

When a function having required arguments is called, the calling program must pass a parameter for each required argument and must pass those parameters in the correct left-to-right order relative to the argument list.

When a function having default arguments is called, the calling program can omit passing parameters to the arguments on the right end of the argument list. However, the parameters that are passed to default arguments must be passed in the correct left-to-right order relative to the argument list.

You are probably seeing the trend here. When a function having keyword arguments is called, the calling program can pass parameters for all, some, or none of the keyword arguments. Furthermore, the parameters that are passed to keyword arguments can be passed in any order. Positional matching is no longer required.

A function can define required arguments, default arguments, keyword arguments, and variable-length arguments (*to be discussed later*) in a variety of combinations. In this module, I will combine required arguments with each of the other types but won't provide examples of combining all four types in the same function.

A function can define one or more keyword arguments in addition to required arguments as shown in [Listing 4](#). Default values are defined for the keyword arguments when the function is defined using the syntax shown in [Listing 4](#). (*I told you earlier that the syntax for defining default arguments is the same as the syntax for defining keyword arguments.*)

The calling program can ignore none, some, or all of the keyword arguments when calling the function. By this I mean that the calling program can simply not pass parameters for keyword arguments. Parameters that are passed for keyword arguments can be passed in any order. In other words, unlike default arguments, the calling program can ignore a keyword argument in the middle of a group of keyword arguments.

As with default arguments, when the calling program ignores keyword arguments, the defined default values for those arguments are used by the code in the body of the function.

The program shown in [Listing 4](#) illustrates a function with one required argument (**listA**) and three keyword arguments (**listB** , **listC** , and **listD**).

Listing 4 . Illustration of keyword arguments.

```
# Illustrates keyword arguments
#-----
-----
```

Listing 4 . Illustration of keyword arguments.

```
def listModifier(listA,listB=["B"],listC=
["C"],listD=["D"]):
    """Illustrates keyword arguments"""
    print("In listModifier")

    listA.append(1.000001)
    print("listA = " + str(listA))
    listB.append(2.000002)
    print("listB = " + str(listB))
    listC.append(3.000003)
    print("listC = " + str(listC))
    listD.append(4.000004)
    print("listD = " + str(listD))

    return
#End function definition

aList = ["ab","cd","ef"]
bList = ["The","old","list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList,listD=dList,listB=bList)
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
```

Listing 4 . Illustration of keyword arguments.

```
print("cList = " + str(cList))  
print("dList = " + str(dList))
```

The code in [Listing 4](#) produces the output shown in [Figure 5](#).

Figure 5 . Output produced by the code in Listing 4.

```
aList = ['ab', 'cd', 'ef']  
bList = ['The', 'old', 'list']  
cList = ['This old house']  
dList = ['is falling down']  
Call listModifier  
In listModifier  
listA = ['ab', 'cd', 'ef', 1.00001]  
listB = ['The', 'old', 'list', 2.00002]  
listC = ['C', 3.00003]  
listD = ['is falling down', 4.00004]  
Back from listModifier  
aList = ['ab', 'cd', 'ef', 1.00001]  
bList = ['The', 'old', 'list', 2.00002]  
cList = ['This old house']  
dList = ['is falling down', 4.00004]
```

As before, the first thing that you should pay attention to is the syntax for defining keyword arguments in the function named **listModifier** in [Listing 4](#). If you compare it with the argument list for the function with the same

name in [Listing 3](#), you will see that there are **exactly the same** . Therefore, the manner in which the arguments are defined does not distinguish default arguments from keyword arguments. The distinguishing factor is the syntax with which the function is called.

The second thing you should pay attention to is the syntax used to call the function named **listModifier** about six lines up from the bottom of [Listing 4](#) . The function call passes three parameters: **aList** , **dList** , and **bList** .

The reference variable named **aList** is passed as the first parameter to satisfy the required argument named **listA** . Note that it is passed using the same syntax as in [Listing 1](#) , [Listing 2](#) , and [Listing 3](#) . Because it is a required argument, it must be passed in the correct order in a positional sense.

The syntax for passing **dList** and **bList** however is significantly different from the previous examples. What you see is something closely akin to an assignment statement. In other words, the parameter named **dList** is *assigned* to the argument named **listD** . Also, the parameter named **bList** is assigned to the argument named **listB** . Nothing is passed and assigned to the argument named **listC** . Furthermore, the two parameters that are passed through assignment to the named arguments are passed in reverse order relative to the definition of those arguments in the function definition.

Hopefully this explanation along with the code in [Listing 4](#) and the output in [Figure 5](#) will tell you what you need to know about using keyword function arguments.

I also recommend that you create a [visualization](#) for the code in [Listing 4](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of keyword arguments.

Variable-length arguments

The program in [Listing 5](#) defines a function named **listModifier** with one required argument (**listA**) and a syntax that supports an arbitrary number of arguments (***wxyz**). *(Note the asterisk, *, immediately to the left of wxyz .)*

The program in [Listing 5](#) illustrates *variable-length arguments* .

Listing 5 . Illustration of variable-length arguments.

```
# Illustrates variable-length arguments
#-----
-----

def listModifier(listA,*wxyz):
    """Illustrates variable-length arguments"""
    print("In listModifier")

    #append a numeric value to the list
    #referenced by required argument listA
    listA.append(1.00001)

    #append increasing numeric values to lists
    #referenced by other parameters
    count = 2
    for ref in wxyz:
        ref.append(1.00001 * count)
        count += 1 #end for loop here

    return #return nothing
#End function definition
```

Listing 5 . Illustration of variable-length arguments.

```
aList = ["ab", "cd", "ef"]
bList = ["The", "old", "list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList, bList, cList, dList)
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))
```

The code in [Listing 5](#) produces the output shown in [Figure 6](#).

Figure 6 . Output produced by the code in Listing 5.

Figure 6 . Output produced by the code in Listing 5.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house', 3.00003]
dList = ['is falling down', 4.00004]
```

The first thing that you should pay attention to is the syntax for defining variable-length arguments in the function named **listModifier** in [Listing 5](#). The syntax consists of an asterisk (*) followed by an arbitrary argument name. As mentioned earlier, the function definition in [Listing 5](#) consists of a required argument (**listA**) followed by the syntax for a variable-length group of arguments (***wxyz**).

In this program, the calling program must pass a parameter for the required argument and can pass an arbitrary number of parameters following the required argument.

The second thing you should pay attention to is the syntax used to call the function named **listModifier** about six lines up from the bottom of [Listing 5](#). The function call passes four parameters: **aList**, **bList**, **cList**, and **dList**, each of which is a reference to a list object.

The reference variable named **aList** is passed as the first parameter to satisfy the required argument named **listA**. Note that it is passed using the same syntax as in [Listing 1](#), [Listing 2](#), [Listing 3](#) and [Listing 4](#). Because it

is a required argument, it must be passed in the correct order in a positional sense.

The remaining three parameters are passed as variable-length arguments. The interpreter wraps them in a tuple and presents the tuple to the code in the body of the function. *(You learned about tuples in the earlier module titled [Itse1359-1100-Indexing and Slicing Tuples](#) and several modules following that one.)*

The code in the body of the function uses the parameter passed as the required argument to append a numeric value to the list referred to by that parameter.

Although a tuple is immutable, the list objects referred to by the elements in the tuple are mutable. A **for** loop in the functions iterates from the beginning to the end of the tuple, extracting the references to the lists and appending an increasing numeric value to the end of each list. This is shown in the last four lines of text in [Figure 6](#).

Note that if the order in which the function processes the arguments is important, the order in which the calling program passes the parameters must match that order because that is the order in which they will be wrapped in the tuple.

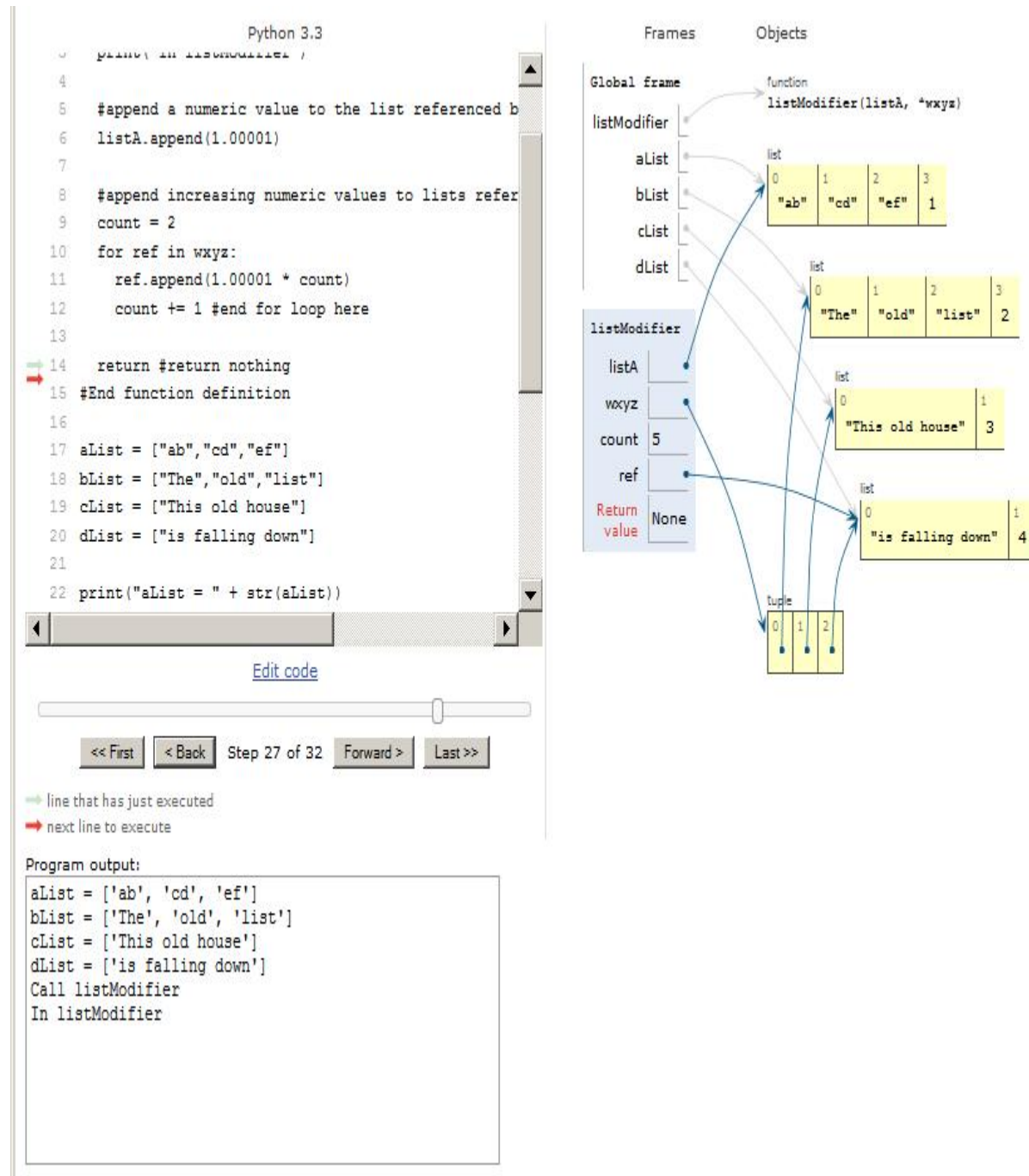
Hopefully this explanation along with the code in [Listing 5](#) and the output in [Figure 6](#) will tell you what you need to know about using variable-length function arguments. I do want to caution again, however, that this module does not provide a complete treatment of function arguments. For a more complete treatment, I will refer you to [The Python Tutorial -- More on Defining Functions](#) and [tutorialspoint -- Python Functions](#).

Visualization of variable-length arguments

[Figure 7](#) shows a [visualization](#) of the code in [Listing 5](#) part of the way through the execution of the program. Note that in order to reduce the amount of vertical space required to publish the [visualization](#), one of the

visualization parameters was changed to *"inline primitives and nested objects [default]."*

Figure 7. Visualization of variable-length arguments.



I recommend that you create a [visualization](#) for the code in [Listing 5](#) and step through the program one instruction at a time. As you do that, pay

attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of variable-length arguments.

Run the programs

I encourage you to copy the code from [Listing 1](#), [Listing 2](#), [Listing 3](#), [Listing 4](#), and [Listing 5](#). Execute the code and confirm that you get the same results as those shown in in this module. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

I also recommend that you create [visualizations](#) for the code in [Listing 1](#) through [Listing 5](#). Step through the programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concepts embodied in those sample programs.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1280-Function Arguments
- File: Itse1359-1280.htm
- Published: 10/26/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1280r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1280-Function Arguments.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1280-Function Arguments* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

Figure 1 . Question 1 program code.

```
def listModifier(listA,listB):
    """Illustrates pass by value or reference"""
    print("In listModifier")

    print("Use incoming parameter to append to
listA")
    listA.append(3.14159)
    print("New listA = " + str(listA))

    print("Assign a new list to listB")
    listB = ["A","new","list"]
    print("New listB = " + str(listB))

    return
#End function definition

#Call the function
print("Create two lists")
aList = ["ab","cd","ef"]
bList = ["The","old","list"]

print("aList = " + str(aList))
print("bList = " + str(bList))

print("Call listModifier")
listModifier(aList,bList)
print("Back from listModifier")
print("aList = " + str(aList))
print("bList = " + str(bList))
```

Figure 2 . Question 1 possible output.

```
Create two lists
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
Call listModifier
In listModifier
Use incoming parameter to append to listA
New listA = ['ab', 'cd', 'ef', 3.14159]
Assign a new list to listB
New listB = ['A', 'new', 'list']
Back from listModifier
aList = ['ab', 'cd', 'ef', 3.14159]
bList = ['A', 'new', 'list']
```

Go to [answer 1](#)

Question 2

True or False? The code in [Figure 4](#) produces the output shown in [Figure 5](#).

Figure 4 . Question 2 program code.

```
def listModifier(listA,listB=["B"],listC=
["C"],listD=["D"]):
```

Figure 4 . Question 2 program code.

```
"""Illustrates default arguments"""
print("In listModifier")

listA.append(1.00001)
print("listA = " + str(listA))
listB.append(2.00002)
print("listB = " + str(listB))
listC.append(3.00003)
print("listC = " + str(listC))
listD.append(4.00004)
print("listD = " + str(listD))

return
#End function definition

aList = ["ab","cd","ef"]
bList = ["The","old","list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList,bList,dList)
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))
```

Figure 5 . Question 2 possible output.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
listA = ['ab', 'cd', 'ef', 1.00001]
listB = ['The', 'old', 'list', 2.00002]
listC = ['C', 3.00003]
listD = ['D', 4.00004]
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house']
dList = ['is falling down']
```

Go to [answer 2](#)

Question 3

True or False? The code in [Figure 7](#) produces the output shown in [Figure 8](#).

Figure 7 . Question 3 program code.

Figure 7 . Question 3 program code.

```
def listModifier(cList,dList=["B"],aList=
["C"],bList=["D"]):
    """Illustrates keyword arguments"""
    print("In listModifier")

    cList.append(1.000001)
    print("cList = " + str(cList))
    dList.append(2.000002)
    print("dList = " + str(dList))
    aList.append(3.000003)
    print("aList = " + str(aList))
    bList.append(4.000004)
    print("bList = " + str(bList))

    return
#End function definition

listV = ["ab","cd","ef"]
listW = ["The","old","list"]
listT = ["This old house"]
listU = ["is falling down"]

print("listV = " + str(listV))
print("listW = " + str(listW))
print("listT = " + str(listT))
print("listU = " + str(listU))

print("Call listModifier")
listModifier(listV,bList=listU,dList=listW)
print("Back from listModifier")

print("listV = " + str(listV))
print("listW = " + str(listW))
```

Figure 7 . Question 3 program code.

```
print("listT = " + str(listT))  
print("listU = " + str(listU))
```

Figure 8 . Question 3 possible output.

```
listV = ['ab', 'cd', 'ef']  
listW = ['The', 'old', 'list']  
listT = ['This old house']  
listU = ['is falling down']  
Call listModifier  
In listModifier  
cList = ['ab', 'cd', 'ef', 1.00001]  
dList = ['The', 'old', 'list', 2.00002]  
aList = ['C', 3.00003]  
bList = ['is falling down', 4.00004]  
Back from listModifier  
listV = ['ab', 'cd', 'ef', 1.00001]  
listW = ['The', 'old', 'list', 2.00002]  
listT = ['This old house']  
listU = ['is falling down', 4.00004]
```

Go to [answer 3](#)

Question 4

True or False? The code in [Figure 9](#) produces the output shown in [Figure 10](#).

Figure 9 . Question 4 program code.

```
def listModifier(listA, *args):  
    print(listA)  
  
    for arg in args:  
        print(arg)  
  
#End function definition  
  
aList = ["ab", "cd", "ef"]  
bList = ["The", "old", "list"]  
cList = [1, 2, 3]  
  
listModifier(aList, bList, cList)
```

Figure 10 . Question 4 possible output.

Figure 10 . Question 4 possible output.

```
['ab', 'cd', 'ef']  
['The', 'old', 'list']  
[1, 2, 3]
```

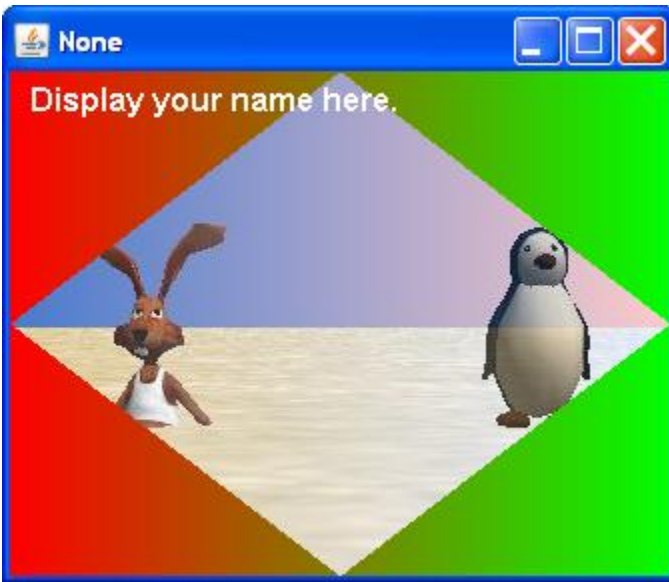
Go to [answer 4](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.
- [Figure 4](#). Question 2 program code.
- [Figure 5](#). Question 2 possible output.
- [Figure 6](#). Question 2 actual output.
- [Figure 7](#). Question 3 program code.
- [Figure 8](#). Question 3 possible output.
- [Figure 9](#). Question 4 program code.
- [Figure 10](#). Question 4 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 4

True.

Go back to [Question 4](#)

Answer 3

True.

Go back to [Question 3](#)

Answer 2

False. The actual output is shown in [Figure 6](#).

Figure 6 . Question 2 actual output.

Figure 6 . Question 2 actual output.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
listA = ['ab', 'cd', 'ef', 1.00001]
listB = ['The', 'old', 'list', 2.00002]
listC = ['is falling down', 3.00003]
listD = ['D', 4.00004]
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house']
dList = ['is falling down', 3.00003]
```

Go back to [Question 2](#)

Answer 1

False. The actual output is shown in [Figure 3](#).

Figure 3 . Question 1 actual output.

Figure 3 . Question 1 actual output.

```
Create two lists
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
Call listModifier
In listModifier
Use incoming parameter to append to listA
New listA = ['ab', 'cd', 'ef', 3.14159]
Assign a new list to listB
New listB = ['A', 'new', 'list']
Back from listModifier
aList = ['ab', 'cd', 'ef', 3.14159]
bList = ['The', 'old', 'list']
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1280r-Review
- File: Itse1359-1280r.htm
- Published: 10/26/14
- Revised: 03/02/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1410-Overview of Python classes

This is the first of several modules that explain classes and objects in Python. This module provides an overview. Future modules will dig deeper into the details of using classes and objects in Python.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [General background information](#)
 - [A class definition is a blueprint for objects](#)
 - [Three important OOP concepts](#)
 - [What is an Object-Oriented Program?](#)
 - [What is encapsulation?](#)
- [Discussion and sample code](#)
 - [A real-world analogy -- a car radio](#)
 - [The ability to store data](#)
 - [The user interface](#)
 - [Modifying the stored data](#)
 - [Responding to a message](#)
 - [Jargon](#)
 - [Where do objects come from?](#)
 - [Program code](#)
 - [Manufacture a three-button radio](#)
 - [Scan for available stations](#)

- [Program the buttons on radio01](#)
 - [Play the three programmed stations](#)
 - [Manufacture another 3-button radio](#)
 - [The class named Radio](#)
 - [A class variable named stations](#)
 - [The `__init__` method](#)
 - [The word `self`](#)
 - [Three more methods](#)
-
- [Visualizing a class definition](#)
 - [Run the program](#)
 - [Complete program listing](#)
 - [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Previous modules have helped you to learn the basics of programming in Python such as syntax, numbers, variables, strings, scripts, lists, tuples, dictionaries, operators, etc.

Previous modules have also helped you to learn about control flow including **if** statements, **while** loops, **for** loops, functions, function arguments, etc.

What you will learn

The time has come for you to learn about *classes* and *objects* . This is the first of several modules that explain classes and objects in Python.

This module provides an overview. Future modules will dig deeper into the details of using classes and objects.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Output from the code in Listing 2.
- [Figure 2](#). Output from the code in Listing 3.
- [Figure 3](#). Output from the code in Listing 4.
- [Figure 4](#). Output from the code in Listing 5.
- [Figure 5](#). Visualizing a class definition.
- [Figure 6](#). Output from the program in Listing 9.

Listings

- [Listing 1](#). Manufacture a three-button radio.
- [Listing 2](#). Scan for available stations.
- [Listing 3](#). Program the buttons on radio01.
- [Listing 4](#). Play the three programmed stations.
- [Listing 5](#). Manufacture another 3-button radio.
- [Listing 6](#). Beginning of the class named Radio.
- [Listing 7](#). The `__init__` method.
- [Listing 8](#). Three more methods.
- [Listing 9](#). Complete program listing.

Preview

This module will concentrate primarily on a discussion of the Python class and objects created using Python classes.

In order to relate object-oriented programming to the real world, a car radio will be used to illustrate and discuss several aspects of software objects. For example, you will learn that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data. You will learn that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

A simple Python program will be presented and explained to illustrate the definition and use of a Python class. This class simulates the manufacture, programming, and use of a car radio.

You will see the definition of a class named **Radio** . You will see how to write code that simulates pressing the *scan* button on the radio to learn about available radio stations in the area. You will see how to write code that simulates the association of a radio button with a particular radio station. You will see how to write code that simulates the pressing of a radio button to play the radio station associated with that button.

You will see how to write code to create new **Radio** objects. You will also see how to save these object's reference in reference variables and how to use those variables to exercise the **Radio** objects.

You will learn some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors. You will learn where objects come from, and you will learn that a class is a set of plans that can be used to construct objects.

You will learn that a Python object is an instance of a class. You will see Python code, used to create two objects from the same class, and then to send messages to each of those objects (*call methods on objects*) .

General background information

A class definition is a blueprint for objects

A class definition is a blueprint (*set of plans*) from which many individual objects can be constructed, created, produced, instantiated, or whatever verb you prefer to use for building something from a set of plans.

An object is a programming construct that encapsulates data and the ability to manipulate that data in a single software entity. The blueprint describes the data contained within and the behavior of objects instantiated according to the class definition.

An object's data is contained in variables defined within the class (*often called member variables, instance variables, data members, attributes, fields, properties, etc.*). The terminology used in the literature often depends on the background of the person writing the document.

Note that unlike Java and C++, once a Python object is instantiated, it can be modified in ways that no longer follow the blueprint of the class through the addition of new data members.

An object's behavior is controlled by methods defined within the class. In Python, methods are functions that are defined within a class definition.

An object is said to have *state* and *behavior* . At any instant in time, the state of an object is determined by the values stored in its variables and its behavior is determined by its methods.

Three important OOP concepts

OOP is an abbreviation for Object-Oriented Programming. Most books on OOP will tell you that in order to understand OOP, you need to understand **the following three concepts** :

- Encapsulation
- Inheritance
- Polymorphism

I will discuss the first two concepts in more detail in this and future modules. As near as I can tell, unlike C++ and Java, Python does not support polymorphism, at least not in any significant way.

C++ and Java support two forms of polymorphism:

- Compile-time polymorphism
- Runtime polymorphism

Both of these depend on the "strongly-typed" nature of C++ and Java. Because Python is a "weakly-typed" (*if typed at all*) programming language, I don't know how to implement either form of polymorphism using Python. (*However, if I am wrong on this, please let me know and I will be happy to learn how to implement polymorphism in Python.*)

Generally, speaking, the concepts in the [above list](#) increase in difficulty going down the list from top to bottom. Therefore, I will begin with encapsulation and work my way down the list in successive modules.

What is an Object-Oriented Program ?

Many authors would answer this question something like the following:

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an object ?

An object is a software construct that encapsulates data, along with the ability to use or modify that data.

What is encapsulation ?

An interesting description of encapsulation was provided in an article by Rocky Lhotka regarding VB.NET. That description reads as follows:

"Encapsulation is the concept that an object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface. The idea is that we can create an interface (Public methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method thus the interface is independent of the implementation."

I like this description, so I won't try to improve on it. I do need to point out that according to [The Python Tutorial](#),

"In fact, nothing in Python makes it possible to enforce data hiding -- it is all based upon convention."

Within that restriction, it is still possible to approximate encapsulation as described by Lhotka using Python provided that appropriate conventions are adhered to.

Discussion and sample code

A real-world analogy -- a car radio

Abstract concepts, such as the concept of an object or encapsulation, can often be best understood by comparing them to real-world analogies. One imperfect, but fairly good analogy to a software object is the radio in your car.

The ability to store data

Your car radio probably has the ability to store data, and to allow you to use and modify that data at will. *(However, you can only use and modify that data through use of the human interface that is provided by the manufacturer of the radio.)* The data that can be stored in your car radio probably includes a list of five or more frequencies that correspond to your favorite radio stations.

Most modern car radios are much more complex than the one that we will use as an analogy in this module.

The user interface

The radio provides a mechanism (*user interface*) that allows you to use the data stored therein. When you press one of the station-selector buttons on the front of the radio, the radio automatically tunes itself to the frequency corresponding to that button. *(In this case, you, the user, are sending a message to the radio object asking it to perform a particular action.)* If you have previously stored a favorite radio station frequency in the storage location corresponding to that button, pressing the button (*sending the message*) will cause the radio station transmitting at that frequency to be heard through the radio's speakers.

If you have not previously stored a favorite frequency in the storage location corresponding to that button, you will probably only hear static. *(That doesn't mean that the radio object failed to respond correctly to the message. It simply means that its response was based on bad data.)*

Modifying the stored data

The human interface also makes it possible for you to store or modify those five or more frequency values. This is done in different ways for different radios. On my car radio, the procedure is:

- Manually tune the radio to the desired frequency
- Press one of the buttons and hold it down for several seconds.

When the radio beeps, I know that the new frequency value has been stored in a storage location that corresponds to that particular button.

What I have done in this process is to send a message to the radio object asking it to *change its state* . The beep that I hear could be interpreted as the radio object returning a value back to me indicating that the mission has been accomplished. (*Alternately, we might say that the radio object sent a message back to me.*)

Responding to a message

We say that an object has changed its state when one or more data values stored in the object have been modified. We also say that when an object responds to a message, it will usually

- perform an action,
- change its state,
- return a value, or
- some combination of the above.

After following this procedure to program a button, when I press that button (*send a message*) , the radio object will automatically tune itself to that frequency.

I live in Austin, TX. If I drive to Dallas and press a button that I have associated with a particular radio station in Austin, I will probably hear static. In that case, I may want to change the frequency value associated with that button. I can follow the same procedure described earlier to *set the frequency value* associated with that button to correspond to one of the radio stations in Dallas. (*Again, I would be sending a message to the radio object asking it to change its state.*)

Jargon

As you can see from the above discussion, the world of OOP is awash with jargon, and the ability to translate the jargon is essential to an understanding of the published material on OOP. Therefore, as we progress through this series of

modules, I will introduce you to some of that jargon and try to help you understand the meaning of the jargon.

Persistence

The ability of your car radio to remember your list of favorite stations is often referred to as persistence. An object that has the ability to store and remember values is often said to have persistence.

State

It is often said that the state of an object at a particular point in time is determined by the values stored in the object. In our analogy, even if we own identical radios, unless the two of us have the same list of favorite radio stations, associated with the same combination of buttons, the state of your radio object at any particular point in time will be different from the state of my radio object.

It is perfectly OK for the two of us to own identical radios and to cause the two radio objects to contain the same list of frequencies. Even if two objects have the same state at the same time, they are still separate and distinct objects. While this is obvious in the real world of car radios, it may not be quite as obvious in the virtual world of computer programming.

Sending a message

A person who speaks in OOP-speak might say that pressing one of the station-selector buttons on the front of the radio sends a message to the radio object, asking it to perform an action (*tune to a particular station*). That person might also say that storing a new frequency that corresponds to a particular button entails sending a message to the radio object asking it to change its state.

Calling a method

Python-speak is a little more specific than general OOP-speak. In Python-speak, we might say that pressing one of the selector buttons on the front of the radio calls a method on the radio object. The behavior of the method is to cause the object to perform an action.

As a practical matter, the physical manifestation of sending a message to an object in Python is to cause that object to execute one of its methods.

Setter methods and getter methods

Similarly, we might say that storing a new frequency that corresponds to a particular button calls a *setter* method on the radio object. *(In an earlier paragraph, I said that I could follow a specific procedure to set the frequency value associated with a button to correspond to one of the radio stations in Dallas. Note the use of the words set and setter in this jargon.)*

Behavior of an object

In addition to state, objects are often also said to have behavior. The overall behavior of an object is determined by the combined behaviors of its individual methods. For example, one of the behaviors exhibited by our radio object is the ability to play the radio station at a particular frequency. When a frequency is selected by pressing a selector button, the radio knows how to translate the radio waves at that frequency into audio waves compatible with our range of hearing, and to send those audio waves out through the speakers. Thus, the radio object behaves in a specific way in response to a message asking it to tune to a particular frequency.

Where do objects come from ?

In order to mass-produce car radios, someone must first create a set of manufacturing plans (*drawings, or blueprints*) for the radio. Once the plans are available, the manufacturing people can produce millions of nearly identical radios.

A Python class definition is a set of plans

The same is true for software objects. In order to create a software object in Python, it is necessary for someone to first create a plan. In Python, we refer to that plan as a ***class*** . The class is defined by a Python programmer. Once the class definition is available, that programmer (*or other programmers*) , can use it to produce large numbers of nearly identical objects.

An instance of a class

If we were standing at the output end of the factory that produces car radios, we might pick up a new radio and say that it is an instance of the plans used to produce the radio. (*Unless they were object-oriented programmers, the people around us might think we were a little odd when they hear us say that.*) However, it is common jargon to refer to a software object as an instance of a class.

To instantiate an object

Furthermore, somewhere along the way, someone turned the word instance into a verb, and it is also common jargon to say that when creating a new object, we are *instantiating* an object.

Program code

As mentioned above, I will explain a program that uses the analogy of a car radio to illustrate several aspects of Python classes and objects.

I will explain this program in fragments. A complete listing of the program is provided in [Listing 9](#) near the end of the module. The output from running the program is shown in [Figure 6](#) . Because this is an overview module, this explanation will gloss over various details. I will revisit and explain many of those details in future modules.

Manufacture a three-button radio

As you will see if you examine [Listing 9](#), this program defines a class named **Radio** . That class can be used to create objects that simulate car radios having three station-selector buttons and a scan button. I will explain that class later. First I will explain how to use the class.

[Listing 1](#) shows the code necessary to create (*instantiate*) a single instance of the **Radio** class and to store that object's reference in a variable named **radio01** . (*In OOP jargon we say that an object is an instance of a class.*) This simulates the manufacturing of the radio and the installation of the radio in a car.

Listing 1 . Manufacture a three-button radio.

```
#Manufacture a 3-button radio  
  
radio01 = Radio()
```

As a practical matter, the code required to instantiate a Python object looks just like the code required to call a function having the same name as the class from which the object is being instantiated. In my opinion, this causes Python to be a little less readable than Java, which uses a special syntax to create an object.

Scan for available stations

Next we want to program the station-selector buttons to cause our favorite radio stations to be played when we press a button. First, however, we will simulate pressing the *scan* button on the radio to learn about the stations that are available in the area.

[Listing 2](#) uses the reference variable named **radio01** to *call a method* named **scan** belonging to the object referred to by **radio01** . As you will see later when we examine the class definition, the **scan** method requires the name of the city as an incoming parameter and returns a reference to a dictionary that relates frequencies to radio station call signs in that city. The returned reference is stored in the variable named **radio01Stations** .

The dictionary is printed in [Listing 2](#) producing the output shown in [Figure 1](#) .

Listing 2 . Scan for available stations.

```
#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio01Stations = radio01.scan("Austin")
print("Available stations in Austin")
print(radio01Stations)
```

As mentioned above, [Figure 1](#) shows the output produced by the code in [Listing 2](#) .

Figure 1 . Output from the code in Listing 2.

Figure 1 . Output from the code in Listing 2.

```
Available stations in Austin  
{93.7: 'KLBJ', 91.7: 'KVRX', 98.1: 'KVET', 95.5:  
'KKMJ'}
```

Program the buttons on radio01

This program simulates a radio having three station-selector buttons. The code in [Listing 3](#) uses the object's reference stored in **radio01** to call a method named **setStationNumber** three times in succession on the object to program each of the three buttons to the frequencies shown. *(Note that the buttons are numbered 1, 2, and 3 instead of 0, 1, and 2.)*

Listing 3 . Program the buttons on radio01.

```
print("Program the buttons")  
radio01.setStationNumber(1,radio01Stations[91.7])  
radio01.setStationNumber(2,radio01Stations[95.5])  
radio01.setStationNumber(3,radio01Stations[98.1])
```

The output produced by the code in [Listing 3](#) is shown in [Figure 2](#).

Figure 2 . Output from the code in Listing 3.

Program the buttons

Play the three programmed stations

Now that our favorite radio stations in the local area (*Austin*) have been programmed into the buttons, we can play any of the three stations simply by turning the radio on and pressing a station-selector button.

[Listing 4](#) calls the **playStation** method three times in succession on the **radio01** object to simulate pressing each of the three buttons. This program doesn't actually produce sound. Instead, it simulates playing a radio station by printing a message identifying the call sign of the station being played.

Listing 4 . Play the three programmed stations.

```
print("Play the three programmed stations")
radio01.playStation(3)
radio01.playStation(2)
radio01.playStation(1)
```

[Figure 3](#) shows the output produced by the code in [Listing 4](#).

Figure 3 . Output from the code in Listing 4.

```
Play the three programmed stations
Playing KVET
Playing KKMJ
Playing KVRX
```

Manufacture another 3-button radio

As I explained earlier, once a class definition is available, it can be used to instantiate any number of objects. This is illustrated in [Listing 5](#), which simulates the manufacturing, programming, and playing of a different radio object.

As you can see in [Listing 5](#), the buttons on this radio are programmed for stations in Dallas suggesting the owner of this radio lives in Dallas.

Listing 5 . Manufacture another 3-button radio.

Listing 5 . Manufacture another 3-button radio.

```
#Manufacture another 3-button radio
radio02 = Radio()

#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio02Stations = radio02.scan("Dallas")
print("Available stations in Dallas")
print(radio02Stations)

print("Program the buttons")
radio02.setStationNumber(1,radio02Stations[91.7])
radio02.setStationNumber(2,radio02Stations[97.9])
radio02.setStationNumber(3,radio02Stations[98.3])

print("Play the three programmed stations")
radio02.playStation(3)
radio02.playStation(2)
radio02.playStation(1)
```

[Figure 4](#) shows the output produced by the code in [Listing 5](#).

Figure 4 . Output from the code in Listing 5.

Figure 4 . Output from the code in Listing 5.

```
Available stations in Dallas
{98.3: 'KNON', 98.7: 'KLUV', 91.7: 'KKXT', 97.9:
'KBFB'}
Program the buttons
Play the three programmed stations
Playing KNON
Playing KBFB
Playing KKXT
```

The class named Radio

The definition of the class named **Radio** begins in [Listing 6](#). The first line of code in a class definition consists of the keyword `class` followed by the name of the class. This is followed by the name of another class in parentheses, which I will explain in a future module on *class inheritance* . As is common in Python, the closing parenthesis is followed by a colon (:) character. The colon is followed by the indented class body.

Listing 6 . Beginning of the class named Radio.

Listing 6 . Beginning of the class named Radio.

```
class Radio(object):  
    #This class provides the plans from which the  
    radio objects are built.  
    stations = {"Austin":  
{91.7:"KVRX", 95.5:"KKMJ", 98.1:"KVET", 93.7:"KLBJ"},  
               "Dallas":  
{98.3:"KNON", 91.7:"KKXT", 97.9:"KBFB", 98.7:"KLUV"}  
}
```

A class variable named stations

The class body in [Listing 6](#) begins with a variable named **stations** that references a dictionary object. The dictionary object contains two nested dictionary objects. One of the nested objects provides information about radio stations in Austin. The other nested object provides information about radio stations in Dallas. This is the source of the radio call sign information for Austin and Dallas shown in [Figure 3](#) and [Figure 4](#).

IMPORTANT: According to [The Python Tutorial -- Class and Instance Variables](#), the variable named **stations** is a *class variable* that is shared by all instances (*objects*) of the class. I will have more to say about *class variables* and *instance variables* in a future module.

The `__init__` method

As I mentioned earlier, when a function is defined inside of a class definition, it is called a method. [Listing 7](#) defines a special method named `__init__`. (Note that two underscore characters are required on each side of the word **init**.)

Listing 7 . The `__init__` method.

```
def __init__(self):  
    self.stationNumber = [0,0,0]
```

According to [The Python Tutorial -- Class Objects](#),

"When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance."

Thus, the `__init__` method can be used to initialize various aspects of a new object when it is instantiated. (*This is roughly analogous to the constructor in other OO languages.*)

The code inside the `__init__` method in [Listing 7](#) creates a new variable named **stationNumber** and initializes it to refer to a three-element list.

According to [The Python Tutorial -- Class and Instance Variables](#), variables that are created in this manner (*inside class methods*) are called *instance variables* and are "for data unique to each instance (object)."

Therefore, the variable named **stations** from [Listing 6](#) is shared among **radio01** and **radio02** in [Listing 9](#). However, **radio01** and **radio02** each has its own copy of the variable named **stationNumber** from [Listing 7](#) and those variables are not shared among objects.

The word **self**

Note the use of the word **self** in two locations in [Listing 7](#). I will explain the use of the word **self** in class definitions in a future module.

Three more methods

[Listing 8](#) defines three more methods. Except for the use of the word **self**, there is nothing unusual about the code in these methods.

Listing 8 . Three more methods.

```
def scan(self,city):  
    return self.stations[city]  
  
def setStationNumber(self,index,station):  
    self.stationNumber[index-1] = station  
  
def playStation(self,index):  
    print("Playing " +  
self.stationNumber[index-1])
```

The scan method

The purpose of the method named **scan** is to simulate pressing the *scan* button on the radio. This method is called on the **Radio** object referred to by **radio01** in [Listing 2](#). It returns a reference to one of the nested dictionary objects in the **stations** dictionary object.

It is called again on a different **Radio** object referred to by **radio02** in [Listing 5](#). It is important to note that even though both objects were instantiated from the same **Radio** class, they are different objects. The only things they share are the **stations** variable, their ancestry, and their overall structure.

*The analogy to a physical car radio breaks down with respect to the class variable named **stations**. Physical car radios don't share any data. You could say that data stored in physical car radios is stored in instance variables only.*

The `setStationNumber` method

The purpose of the **setStationNumber** method is to simulate programming the station-selector buttons on a **Radio** object. It is called three times, once for each button, on the object referred to by **radio01** in [Listing 3](#). It is also called three times on the different **Radio** object referred to by **radio02** in [Listing 5](#).

The `playStation` method

The purpose of the **playStation** method is to simulate pressing a station-selector button on the radio to play the radio station that has been programmed into that button. It is called three times on the **Radio** object referred to by **radio01** in [Listing 4](#). It is also called three times on the different **Radio** object referred to by **radio02** in [Listing 5](#).

Visualizing a class definition

[Figure 5](#) shows a [visualization](#) of the class definition in [Listing 9](#).

Figure 5. Visualizing a class definition.

The screenshot displays a Python 3.3 IDE with a class definition for `Radio` and its object representation.

Class Definition (Radio):

```

1 class Radio(object):
2     #This class provides the plans from which the radio ob
3     stations = [{"Austin":(91.7:"KVRX",98.5:"KQMJ",98.1:"KV
4         "Dallas":(98.3:"KQON",91.7:"KQXT",97.9:"KB
5     ]
6
7     def __init__(self):
8         self.stationNumber = [0,0,0]
9
10    def scan(self,city):
11        return self.stations[city]
12
13    def setStationNumber(self,index,station):
14        self.stationNumber[index-1] = station
15
16    def playStation(self,index):
17        print("Playing " + self.stationNumber[index-1])
18
19    #Manufacture a 3-button radio

```

Object Representation (Radio class):

Attribute	Value																					
<code>__init__</code>	function <code>__init__(self)</code>																					
<code>playStation</code>	function <code>playStation(self, index)</code>																					
<code>scan</code>	function <code>scan(self, city)</code>																					
<code>setStationNumber</code>	function <code>setStationNumber(self, index, station)</code>																					
<code>stations</code>	dict <table border="1"> <thead> <tr> <th>City</th> <th>Frequency</th> <th>Call Sign</th> </tr> </thead> <tbody> <tr> <td rowspan="4">Dallas</td> <td>97.9</td> <td>KBFB</td> </tr> <tr> <td>98.3</td> <td>KQON</td> </tr> <tr> <td>91.7</td> <td>KQXT</td> </tr> <tr> <td>98.7</td> <td>KLUJ</td> </tr> <tr> <td rowspan="4">Austin</td> <td>98.1</td> <td>KVET</td> </tr> <tr> <td>91.7</td> <td>KVRX</td> </tr> <tr> <td>93.7</td> <td>KLBJ</td> </tr> <tr> <td>95.5</td> <td>KQMJ</td> </tr> </tbody> </table>	City	Frequency	Call Sign	Dallas	97.9	KBFB	98.3	KQON	91.7	KQXT	98.7	KLUJ	Austin	98.1	KVET	91.7	KVRX	93.7	KLBJ	95.5	KQMJ
City	Frequency	Call Sign																				
Dallas	97.9	KBFB																				
	98.3	KQON																				
	91.7	KQXT																				
	98.7	KLUJ																				
Austin	98.1	KVET																				
	91.7	KVRX																				
	93.7	KLBJ																				
	95.5	KQMJ																				

I recommend that you create a [visualization](#) for the code in [Listing 9](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of classes and objects in Python.

Run the program

I also encourage you to copy the code from [Listing 9](#). Execute the code and confirm that you get the same results as those shown in [Figure 6](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is provided in [Listing 9](#) below.

Listing 9 . Complete program listing.

```

# This program simulates the manufacture,
# programming, and use of a pair of
# three-button car radios.
#
#-----
-----
class Radio(object):
    #This class provides the plans from which the
    radio objects are built.
    stations = {"Austin":
{91.7:"KVRX",95.5:"KKMJ",98.1:"KVET",93.7:"KLBJ"},
"Dallas":
{98.3:"KNON",91.7:"KKXT",97.9:"KBFB",98.7:"KLUV"}
}

    def __init__(self):
        self.stationNumber = [0,0,0]

    def scan(self,city):
        return self.stations[city]

    def setStationNumber(self,index,station):
        self.stationNumber[index-1] = station

    def playStation(self,index):
        print("Playing " + self.stationNumber[index-
1])

#Manufacture a 3-button radio
radio01 = Radio()

#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio01Stations = radio01.scan("Austin")
print("Available stations in Austin")
print(radio01Stations)

print("Program the buttons")

```

```
radio01.setStationNumber(1,radio01Stations[91.7])
radio01.setStationNumber(2,radio01Stations[95.5])
radio01.setStationNumber(3,radio01Stations[98.1])

print("Play the three programmed stations")
radio01.playStation(3)
radio01.playStation(2)
radio01.playStation(1)

#Manufacture another 3-button radio
radio02 = Radio()

#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio02Stations = radio02.scan("Dallas")
print("Available stations in Dallas")
print(radio02Stations)

print("Program the buttons")
radio02.setStationNumber(1,radio02Stations[91.7])
radio02.setStationNumber(2,radio02Stations[97.9])
radio02.setStationNumber(3,radio02Stations[98.3])

print("Play the three programmed stations")
radio02.playStation(3)
radio02.playStation(2)
radio02.playStation(1)
```

[Figure 6](#) shows the output produced by the code in [Listing 9](#).

Figure 6 . Output from the program in Listing 9.

Figure 6 . Output from the program in Listing 9.

```
Available stations in Austin
{93.7: 'KLBJ', 91.7: 'KVRX', 98.1: 'KVET', 95.5:
'KKMJ'}
Program the buttons
Play the three programmed stations
Playing KVET
Playing KKMJ
Playing KVRX
Available stations in Dallas
{98.3: 'KNON', 98.7: 'KLUV', 91.7: 'KKXT', 97.9:
'KBFB'}
Program the buttons
Play the three programmed stations
Playing KNON
Playing KBFB
Playing KKXT
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1410-Overview of Python classes
- File: Itse1359-1410.htm
- Published: 10/27/14
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1410r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1410-Overview of Python classes.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1410-Overview of Python classes* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#) except that the order of the items in the dictionaries may be different from one run to the next.

Figure 1 . Question 1 program code.

```
class MyNewClass(object):
    myDict = {"A":
{100:"M",200:"N",300:"P",400:"Q"},
            "B":
{500:"S",600:"R",700:"U",800:"T"}
            }

    def __init__(self):
        self.itemNumber = [0,0,0]

    def scan(self,group):
        return self.myDict[group]

    def setItemNumber(self,index,val):
        self.itemNumber[index-1] = val

    def displayItem(self,index):
        print("Showing " +
self.itemNumber[index-1])
```

Figure 1 . Question 1 program code.

```
group01 = MyNewClass()

group01Items = group01.scan("A")
print("Available items in 'A'")
print(group01Items)

group01.setItemNumber(1,group01Items[200])
group01.setItemNumber(2,group01Items[100])
group01.setItemNumber(3,group01Items[300])

group01.displayItem(3)
group01.displayItem(2)
group01.displayItem(1)

group02 = MyNewClass()

group02Items = group02.scan("B")
print("Available items in 'B'")
print(group02Items)

group02.setItemNumber(1,group02Items[600])
group02.setItemNumber(2,group02Items[700])
group02.setItemNumber(3,group02Items[500])

group02.displayItem(3)
group02.displayItem(2)
group02.displayItem(1)
```

Figure 2 . Question 1 possible output.

```
Available items in 'A'
{200: 'N', 300: 'P', 400: 'Q', 100: 'M'}
Showing P
Showing M
Showing N
Available items in 'B'
{600: 'R', 700: 'U', 500: 'S', 800: 'T'}
Showing S
Showing U
Showing R
```

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1410r-Review
- File: Itse1359-1410r-Review.htm
- Published: 10/27/14
- Revised: 03/02/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1420-Understanding self

This module explains the use of the word self in class definitions.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Multiple objects from the same class](#)
 - [The class named Radio](#)
 - [Class methods versus instance methods](#)
- [Complete program listing](#)
- [Visualizing the program](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

In an earlier module, you learned how to define a Python class, how to instantiate objects of that class, and how to call methods on those objects.

What you will learn

The class-definition code in the earlier module included the word **self** in several different locations. I promised that I would revisit the class definition and explain the use of the word **self** . That is the purpose of this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Visualizing the program.
- [Figure 2](#). Output from the code in Listing 5.

Listings

- [Listing 1](#). Multiple objects from the same class.
- [Listing 2](#). Call the playStation method on an object.
- [Listing 3](#). Abbreviated version of the class named Radio.
- [Listing 4](#). Using self to access an instance variable.
- [Listing 5](#). Complete program listing.

Preview

[Listing 5](#) shows a complete listing of a scaled down version of the radio program from an earlier module. *(The program was scaled down simply to reduce the size and to make the presentation more manageable.)* [Figure 2](#) shows the output produced by the code in [Listing 5](#).

The program in [Listing 5](#) will be used to explain the use of the word **self** in a Python class definition.

Discussion and sample code

With the exception of the following items, you should already understand the code in [Listing 5](#).

- The use of the word **self**
- The difference between *class variables* and *instance variables* .
- The word **object** in the parentheses in the first line of the class definition.

I will use fragments of code from the program in [Listing 5](#) to explain the use of the word **self** in this module.

I will explain the difference between class variables and instance variables as well as the use of the word **object** in future modules.

Multiple objects from the same class

You learned in the earlier module that you can instantiate one or more independent objects from the same class and call methods on those objects. This is illustrated by the abbreviated code in [Listing 1](#). *(Note that comments and code were omitted from [Listing 1](#) to make it easier to discuss the code of interest.)*

Listing 1 . Multiple objects from the same class.

```
#Manufacture a 1-button radio
radio01 = Radio()
## Code omitted for brevity

radio01.playStation(1)
## Code omitted for brevity

#Manufacture another 1-button radio
radio02 = Radio()
## Code omitted for brevity

radio02.playStation(1)
```

The code in [Listing 1](#) instantiates two objects of the class **Radio** . References to those objects are stored in the variables named **radio01** and **radio02** .

Once a variable contains a reference to an object, the name of that variable can be used along with the ***dot operator*** to call a method on that specific object.

The code in [Listing 1](#) uses the references to the two objects to call a method named **playStation** on each of the objects. Even though the name of the method is the same in both cases and the value of the parameter passed to the method is the same in both cases, the result of calling the method on one object is different from the result of calling the method on the other object. This is because the data stored in one object is different from the data stored in the other object.

The behavior of the method is the same in both cases. However, because the method operates on different data while executing each call, the results are different. This is evidenced by the output shown in [Figure 2](#).

Each object occupies its own chunk of memory and stores its instance variables within that chunk of memory. However (*probably for reasons of economy*) , each object does not have its own copy of the methods defined in the class from which the object is instantiated. Instead, all objects instantiated from a given class share a common copy of each method defined in the class. This is where the word **self** comes into play. The word **self** is used to cause the program to behave *as if* each object has its own copy of each method.

Consider the variables named **radio01** and **radio02** in [Listing 1](#). Each of these variables contains a value that is returned by the process of instantiating an object. (*I frequently refer to this value as a reference to the object.*) The value stored in each of these variables identifies the chunk of memory occupied by one object. (*The manner in which this identification is accomplished probably varies among different implementations of Python such as CPython, Jython, PyPy, etc.*)

In any event, when you execute a statement that calls a common method such as the one shown in [Listing 2](#), the method must have a way of identifying the chunk of memory belonging to the object whose reference was used to **make the call** .

Listing 2 . Call the playStation method on an object.

```
radio02.playStation(1)
```

In other words, in the case shown in [Listing 2](#), the method must be able to find and use the chunk of memory identified by the contents of **radio02** as

opposed to the chunk of memory identified by the contents of **radio01** . That is what **self** is all about.

The class named Radio

[Listing 3](#) shows an abbreviated version of the class named **Radio** .

Listing 3 . Abbreviated version of the class named Radio.

```
class Radio(object):  
    ## Code omitted for brevity  
  
    def __init__(self):  
        self.stationNumber = [0,0,0]  
  
    ## Code omitted for brevity  
  
    def playStation(self,index):  
        print("Playing " +  
self.stationNumber[index-1])
```

Methods are often defined with a special argument as the first argument in the argument list. This argument is named **self** in the methods of [Listing 3](#) . Here is some of what [The Python Tutorial -- Random Remarks](#) has to say about **self** .

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

Therefore, even though it is possible to give the first argument a different name, I will stick with the name **`self`** in this module.

The first argument in a function's argument list is special because:

- When an object's method is called using the syntax shown in [Listing 2](#), the first incoming parameter to the method is an identification of the chunk of memory belonging to the object that was used to make the call.
- The client code that makes the call is not responsible for including that information in the argument list when the call is made. The interpreter takes care of that task automatically.

For example, the **`playStation`** method shown in [Listing 3](#) shows two required positional arguments including the argument named **`self`**. The call to the **`playStation`** method shown in [Listing 2](#) passes only one parameter. The argument identified as **`self`** in [Listing 3](#) is automatically constructed and passed to the method by the Python interpreter.

When you write code in the body of the method, you can use the value of **`self`** along with the dot operator to access variables and methods belonging to the object using code such as that shown in [Listing 4](#).

Listing 4 . Using self to access an instance variable.

```
print("Playing " +  
self.stationNumber[index-1])
```

As near as I have been able to determine, references to an object's instance variables or methods by code within the body of a method that is defined in that object's class must qualify those references with **self** (*or whatever name you choose to give to the method's first argument*) using the dot operator.

That restriction does not apply to class variables. I will have more to say about that in a future module. There are probably other exceptions as well having to do with communications between methods in different objects or methods in one object accessing variables in a different object.

Class methods versus instance methods

In other OOP environments, we would probably refer to the method named **playStation** in [Listing 3](#) as an **instance method** . The method is called using an object's reference as shown in [Listing 2](#) . That object's reference is passed into the method's **self** parameter as discussed above.

It is also possible to call a method in a class using the name of the class in place of an object's reference. In that case, an object's reference is not automatically passed to a **self** parameter. In fact the method does not even need to define a **self** parameter. In other programming environments, we would probably refer to such a method as a **class method** . Class methods are typically used to perform an action that is independent of any specific object, such as computing the square root of a number for example.

Complete program listing

A complete listing of the code for the program discussed in this module is provided in [Listing 5](#).

Listing 5 . Complete program listing.

```
# This program simulates the manufacture,
# programming, and use of a pair of
# one-button car radios. The purpose is to explain
# the use of the word self.
#
#-----
-----
class Radio(object):
    #This class provides the plans from which the
    radio objects are built.
    stations = {"Austin":{91.7:"KVRX"},
               "Dallas":{98.3:"KNON"}}
    }

    def __init__(self):
        self.stationNumber = [0,0,0]

    def scan(self,city):
        return self.stations[city]

    def setStationNumber(self,index,station):
        self.stationNumber[index-1] = station

    def playStation(self,index):
        print("Playing " +
self.stationNumber[index-1])

#Manufacture a 1-button radio
radio01 = Radio()
```

```
#Program the button
#First scan for available stations
radio01Stations = radio01.scan("Austin")
print("Available stations in Austin")
print(radio01Stations)

print("Program the button")
radio01.setStationNumber(1,radio01Stations[91.7])

print("Play the  programmed station")
radio01.playStation(1)

#Manufacture another 1-button radio
radio02 = Radio()

#Program the button
#First scan for available stations
radio02Stations = radio02.scan("Dallas")
print("Available stations in Dallas")
print(radio02Stations)

print("Program the button")
radio02.setStationNumber(1,radio02Stations[98.3])

print("Play the  programmed station")
radio02.playStation(1)
```

[Figure 2](#) shows the output produced by the code in [Listing 5](#).

Figure 2 . Output from the code in Listing 5.

Figure 2 . Output from the code in Listing 5.

```
Available stations in Austin
{91.7: 'KVRX'}
Program the button
Play the programmed station
Playing KVRX
Available stations in Dallas
{98.3: 'KNON'}
Program the button
Play the programmed station
Playing KNON
```

Visualizing the program

[Figure 1](#) shows a [visualization](#) of the code shown in [Listing 5](#).

Figure 1. Visualizing the program.

```

Python 3.3
49 radio01.setStationNumber(1,radio01Stations[91.7])
20
31 print("Play the programmed station")
32 radio01.playStation(1)
33
34
35 #Manufacture another 1-button radio
36 radio02 = Radio()
37
38 #Program the button
39 #First scan for available stations
40 radio02Stations = radio02.scan("Dallas")
41 print("Available stations in Dallas")
42 print(radio02Stations)
43
44 print("Program the button")
45 radio02.setStationNumber(1,radio02Stations[98.3])
46
47 print("Play the programmed station")
48 radio02.playStation(1)

```

[Edit code](#)

<< First < Back Program terminated Forward > Last >>

→ line that has just executed
 → next line to execute

Program output:

```

Available stations in Austin
{91.7: 'KVRX'}
Program the button
Play the programmed station
Playing KVRX
Available stations in Dallas
{98.3: 'KNON'}
Program the button
Play the programmed station
Playing KNON

```

Frames

Global frame

- Radio
- radio01
- radio01Stations
- radio02
- radio02Stations

Objects

Radio class

hide attributes

__init__	function	__init__(self)
playStation	function	playStation(self, index)
scan	function	scan(self, city)
setStationNumber	function	setStationNumber(self, index, station)
stations	dict	{ "Austin": {91.7: 'KVRX'}, "Dallas": {98.3: 'KNON'} }

Radio Instance

stationNumber	list		
	0	1	2
	"KVRX"	0	0

Radio Instance

stationNumber	list		
	0	1	2
	"KNON"	0	0

dict

91.7 "KVRX"

dict

98.3 "KNON"

I recommend that you create a [visualization](#) for the code in [Listing 5](#) and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of classes and objects as well as the use of *self* in Python.

Run the program

I also encourage you to copy the code from [Listing 5](#). Execute the code and confirm that you get the same results as those shown in [Figure 2](#).

Experiment with the code, making changes, and observing the results of your changes. For example, change all occurrences of the word **self** to the word **this** to see what happens. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1420-Understanding self
- File: Itse1359-1420.htm
- Published: 10/27/14
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available

on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1420r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1420-Understanding self.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1420-Understanding self*.

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#) except that the order of the items in the dictionary may be different from one run to the next.

Figure 1 . Question 1 program code.

```
class MyNewClass(object):
    myDict = {"A":
{100:"M",200:"N",300:"P",400:"Q"},
            "B":
{500:"S",600:"R",700:"U",800:"T"}
            }

    def __init__(thisObject):
        thisObject.itemNumber = [0,0,0]

    def scan(thisObject,group):
        return thisObject.myDict[group]

    def setItemNumber(thisObject,index,val):
        thisObject.itemNumber[index-1] = val

    def displayItem(thisObject,index):
        print("Showing " +
thisObject.itemNumber[index-1])
```

Figure 1 . Question 1 program code.

```
group01 = MyNewClass()

group01Items = group01.scan("A")
print("Available items in 'A'")
print(group01Items)

group01.setItemNumber(1,group01Items[200])
group01.setItemNumber(2,group01Items[100])
group01.setItemNumber(3,group01Items[300])

group01.displayItem(3)
group01.displayItem(2)
group01.displayItem(1)
```

Figure 2 . Question 1 possible output.

```
Available items in 'A'
{200: 'N', 300: 'P', 400: 'Q', 100: 'M'}
Showing P
Showing M
Showing N
```

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

True.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1420r-Review
- File: Itse1359-1420r.htm
- Published: 10/27/14
- Revised: 03/02/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Testing copy of Itse1359-1430-Instance Variables
For testing. Do not use.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Empty objects](#)
 - [Instantiate two objects of the empty class](#)
 - [Would be almost worthless in Java or C++](#)
 - [Add an instance variable to one object](#)
 - [Try to display the same instance variable in the other object](#)
 - [Visualize the code in Listing 9](#)
 - [Object can modify itself at runtime](#)
 - [The definition of a class named TestClass](#)
 - [Add an instance variable to one object and display it](#)
 - [Visualize the code in Listing 10](#)
 - [The method named `init`](#)
 - [A class with an `init` method](#)
 - [Instantiate and display two objects](#)
 - [The incoming parameter to the `init` method](#)
 - [Display first instance variable from both objects](#)
 - [The remainder of the program](#)

- [Visualize the code in Listing 11](#)
- [Run the program](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Itse1359-1440-Class Variables

I promised in an earlier module that I would revisit and explain the difference between class variables and instance variables in classes and objects. An earlier module explained instance variables in detail. You will learn some of the details regarding class variables in this module.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Late modification of class](#)
 - [A simple class definition](#)
 - [Instantiate and display two objects](#)
 - [Add a class variable directly](#)
 - [Add a class variable via an object](#)
 - [Display both class variables via one object](#)
 - [Display both class variables via the other object](#)
 - [Typical class variable usage](#)
 - [The class definition](#)
 - [Instantiate two objects from the class](#)
 - [Modify class variable using object](#)
 - [Modify the class variable using the class name](#)
 - [A word of caution is in order](#)
 - [Shadowing or hiding a class variable](#)

- [Class definition with a shadowing method](#)
 - [Instantiate and display classVar in two objects](#)
 - [Modify classVar in one object](#)
 - [Modify the contents of the class variable](#)
- [Run the programs](#)
 - [Visualize the programs](#)
 - [Complete program listings](#)
 - [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Earlier modules provided

- an overview of classes and objects in Python,
- an explanation of the **self** word in Python class definitions, and
- a detailed explanation of instance variables in Python class definitions.

You have also learned that although other OO languages such as Java and C++ use terminology that is common with Python terminology, such as *class* , *object* , *class variable* , and *instance variable* , those terms have significantly different meanings in Python than they do in many other OO languages.

What you will learn

I promised in an earlier module that I would revisit and explain the difference between *class variables* and *instance variables* in classes and

objects. An earlier module explained instance variables in detail. You will learn some of the details regarding class variables in this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Output from the code in Listing 2.
- [Figure 2](#). Output of the code in Listing 5.
- [Figure 3](#). Output from the code in Listing 7.
- [Figure 4](#). Output from the code in Listing 8.
- [Figure 5](#). Output from the code in Listing 9.
- [Figure 6](#). Output from the code in Listing 10.
- [Figure 7](#). Output from the code in Listing 12.
- [Figure 8](#). Output from the code in Listing 13.
- [Figure 9](#). Output from the code in Listing 14.
- [Figure 10](#). Output from the code in Listing 15.
- [Figure 11](#). Output from the code in Listing 16.
- [Figure 12](#). Output from the code in Listing 17.

Listings

- [Listing 1](#). A simple class definition.

- [Listing 2](#). Instantiate and display two objects.
- [Listing 3](#). Add a class variable directly.
- [Listing 4](#). Add a class variable via an object.
- [Listing 5](#). Display both class variables via one object.
- [Listing 6](#). Display both class variables via the other object.
- [Listing 7](#). The class definition.
- [Listing 8](#). Instantiate two objects from the class.
- [Listing 9](#). Modify class variable using object.
- [Listing 10](#). Modify the class variable using the class name.
- [Listing 11](#). Class definition with a shadowing method.
- [Listing 12](#). Instantiate and display classVar in two objects.
- [Listing 13](#). Modify classVar in one object.
- [Listing 14](#). Modify the contents of the class variable.
- [Listing 15](#). Complete program listing.
- [Listing 16](#). Complete program listing.
- [Listing 17](#). Complete program listing.

General background information

In Java and C++, once you define and compile a class, that class, (*which is the blueprint for an object*) , can only be modified by recompiling the class. In other words, once compiled, a class is intended to be stable and cannot be modified at runtime. (*At lease I don't know how to modify a Java class definition at runtime.*)

Also in Java and C++, once you instantiate an object from a class, you cannot modify the structure of the object. You can modify the values of the data stored in the object but the structure of the object is stable and cannot be modified at runtime. (*At least I don't know how to modify the structure of a Java object at runtime.*) Among other things, you cannot add new variables to an existing object.

Along that line, the structure of an object in Java and C++ is explicitly tied to the blueprint provided by the class from which it was instantiated. There is a fixed relationship between the object and the class from which it was instantiated.

None of that is true in Python. You learned in an earlier module that you can modify the structure of a Python object at runtime after it is instantiated. By that I mean that you can add new instance variables to an object such that the structure of the object no longer matches the blueprint provided by the class from which it was instantiated.

You will learn in this module that you can modify the blueprint provided by the class at runtime after the class has been used to instantiate one or more objects.

Discussion and sample code

I will discuss three different programs in this module. A complete listing of the first program is provided in [Listing 15](#). The output from that program is shown in [Figure 10](#).

I will break each program down and discuss it in fragments. The first fragment is shown in [Listing 1](#).

Late modification of class

Let me begin by saying that this is probably not how the designers of Python intended for classes and objects to be used. However, if you are going to program using classes and objects in Python, you need to understand them.

A simple class definition

[Listing 1](#) shows a simple Python class definition for a class named **TestClass**. This class definition does not define any class variables and does not define any instance variables. It does define a method named **addClassVar**, which I will discuss in more detail later.

Listing 1 . A simple class definition.

```
class TestClass(object):  
    def addClassVar(self,data):  
        TestClass.classVar02 = data
```

An object instantiated from the class in [Listing 1](#) will not contain any data -
- no class variables and no instance variables.

Instantiate and display two objects

The code in [Listing 2](#) instantiates and prints two objects of the class named **TestClass** . Neither object contains any data and neither class has access to any data.

Listing 2 . Instantiate and display two objects.

```
print("Instantiate and display two objects of  
TestClass")  
ref01 = TestClass()  
print(ref01)  
ref02 = TestClass()  
print(ref02)
```

[Figure 1](#) shows the output from the code in [Listing 2](#). There is nothing new here. You saw code like this in an earlier module.

Figure 1 . Output from the code in Listing 2.

```
Instantiate and display two objects of  
TestClass  
<__main__.TestClass object at 0x02152930>  
<__main__.TestClass object at 0x02152E70>
```

Add a class variable directly

The code in [Listing 3](#) accesses the class named **TestClass** directly by name and adds a class variable named **classVar01** to the class.

Listing 3 . Add a class variable directly.

```
print("Add a class variable directly.")  
TestClass.classVar01 = "ABCD"
```

The interesting question is what impact if any does this have on the two objects that have already been instantiated from the class? I will defer a discussion of this question until after we see the impact.

Add a class variable via an object

The code in [Listing 4](#) calls the method named **addClassVariable** that is defined in the class shown in [Listing 1](#) to add another class variable named **classVar02** to the class named **TestClass** .

Listing 4 . Add a class variable via an object.

```
print("Add a class variable via an object.")
ref01.addClassVar("1234")
```

The purpose of including this code in the program is simply to show that an object can be used to modify the class from which it was instantiated.

Display both class variables via one object

The code in [Listing 5](#) uses the reference to one of the objects, **ref01** , to access and print the values of the two new class variables named **classVar01** and **classVar02** . That brings us back to the question of what impact if any does the addition of two new class variables have on the two objects that have already been instantiated from the class?

Listing 5 . Display both class variables via one object.

```
print("Display both class variables via one  
object")  
print("ref01: " + ref01.classVar01)  
print("ref01: " + ref01.classVar02)
```

Although I am unable to locate a reference to verify this, it is my understanding that when program code references an attribute such as **classVar01** using a reference to an object such as **ref01** , the runtime system first searches the object to see if it contains an attribute with that name. If it doesn't find an attribute with that name in the object, it then searches the class from which the object was instantiated.

The object referred to by **ref01** does not contain an attribute named **classVar01** , but the class named **TestClass** does now contain an attribute with that name (*even though it was not there when the object was instantiated*) .

If both the object and the class contain attributes with the same name, the attribute in the object will shadow or hide the attribute in the class. I will demonstrate the impact of this later.

If the search pattern described [above](#) is correct, we would expect the code in [Listing 5](#) to display the values contained in each of the two new class variables belonging to the class named **TestClass** . [Figure 2](#) shows the output produced by the code in [Listing 5](#) and verifies that the search pattern described [above](#) is probably correct. The values stored in the new class variables are displayed in [Figure 2](#) by the code in [Listing 5](#).

Figure 2 . Output of the code in Listing 5.

```
Display both class variables via one object  
ref01: ABCD  
ref01: 1234
```

Display both class variables via the other object

The code in [Listing 6](#) along with the output shown in [Figure 10](#) shows unsurprisingly that the other object that was instantiated from **TestClass** can be used to access and display the same contents of the same two class variables.

Listing 6 . Display both class variables via the other object.

```
print("Display both class variables via the  
other object")  
print("ref02: " + ref02.classVar01)  
print("ref02: " + ref02.classVar02)
```

This is because class variables belonging to a class are shared among all objects instantiated from that class unless shadowed by an instance variable having the same name in an object of the class. This will be demonstrated in the programs that follow.

Typical class variable usage

[Listing 16](#) shows a program that probably comes closer to the intended use of class variables than the code in the earlier program shown in [Listing 15](#). The output from the code shown in [Listing 16](#) is shown in [Figure 11](#).

This program demonstrates that class variables are shared among the objects instantiated from a class. It also demonstrates that changes to class variables in that class impact all of the objects instantiated from the class.

The class definition

[Listing 7](#) shows the definition of the class for this program including the creation and initialization a class variable named **classVar** . The class definition also includes a couple of methods that I will discuss later.

Listing 7 . The class definition.

Listing 7 . The class definition.

```
class TestClass(object):  
    classVar = [1,2,3,4]  
  
    def modifyClassVar01(self):  
        self.classVar.append("a")  
  
    def printClassVar01(self):  
        print(str(self)[10:40] + ": " +  
str(self.classVar))  
  
print("1  Display class var using class: " +  
str(TestClass.classVar))
```

[Listing 7](#) also contains a print statement that prints the value of the variable named **classVar** by using the name of the class, **TestClass** , to access the variable. The output is shown in [Figure 3](#).

The output shown in [Figure 3](#) faithfully reproduces the initial value of the class variable shown in the second line of text in [Listing 7](#).

Figure 3 . Output from the code in Listing 7.

```
1 Display class var using class: [1, 2, 3, 4]
```

Instantiate two objects from the class

[Listing 8](#) instantiates two separate and distinct objects from the class named **TestClass** . Those objects' references are stored in the variables named **ref01** and **ref02** .

Listing 8 . Instantiate two objects from the class.

```
print("2  Instantiate two objects")
ref01 = TestClass()
ref02 = TestClass()

print("3  Display class var using objects")
ref01.printClassVar01()
ref02.printClassVar01()
```

[Listing 8](#) also uses the variables named **ref01** and **ref02** to call the **printClassVar01** method shown in [Listing 7](#) on each object. This method concatenates and prints a portion of the object identifier (*self*) with the contents of **classVar** (*as seen by the object*) .

[Figure 4](#) shows the output produced by the code in [Listing 8](#) . As you can tell from [Figure 4](#) , each object sees **classVar** as containing the initial value from [Listing 7](#) .

Figure 4 . Output from the code in Listing 8.

```
2 Instantiate two objects
3 Display class var using objects
TestClass object at 0x02091DD0: [1, 2, 3, 4]
TestClass object at 0x0218C710: [1, 2, 3, 4]
```

Modify class variable using object

The code in [Listing 9](#) uses the reference to a single object (*ref01*) to call the method named **modifyClassVar01** (see [Listing 7](#)) on that object, which appends the string "a" onto the list referred to by the class variable named **classVar** . *(As an aside, note that the code in the method uses the word **self** to access the class variable.)*

Listing 9 . Modify class variable using object.

```
print("4  Modify class var using object")
ref01.modifyClassVar01()

print("5  Display class var using objects")
ref01.printClassVar01()
ref02.printClassVar01()
```

After causing the string "a" to be appended to the list, [Listing 9](#) uses the variables named **ref01** and **ref02** to call the **printClassVar01** method on each object. This causes each object to print the list as it sees it. The results are shown in [Figure 5](#).

Figure 5 . Output from the code in Listing 9.

```
4 Modify class var using object
5 Display class var using objects
TestClass object at 0x02091DD0: [1, 2, 3, 4,
'a']
TestClass object at 0x0218C710: [1, 2, 3, 4,
'a']
```

Even though the contents of the list were modified only by code executed in **ref01** , the results are seen by both objects. This is because the class variable is shared among all objects instantiated from the class.

Modify the class variable using the class name

I could have stopped at that point and stated "*case closed*" . However, I decided to add a few more lines of code to illustrate that it doesn't matter how a class variable is modified, it still impacts all objects instantiated from the class because the class variable is shared among all objects instantiated from the class.

The code in [Listing 10](#) uses the name of the class to access and modify the list referred to by the class variable named **classVar** in the class named

TestClass . The boolean value **True** is appended to the list by the code in [Listing 10](#).

Listing 10 . Modify the class variable using the class name.

```
print("6  Modify class var using class")
TestClass.classVar.append(True)

print("7  Display class var using objects")
ref01.printClassVar01()
ref02.printClassVar01()

print("8  Display class var using class: " +
      str(TestClass.classVar))
```

After that, the code in [Listing 10](#) causes each object to print the list as it see it. To seal the deal, the code in [Listing 10](#) uses the name of the class to access and print the value stored in the class variable. The result is shown in [Figure 6](#).

Figure 6 . Output from the code in Listing 10.

Figure 6 . Output from the code in Listing 10.

```
6 Modify class var using class
7 Display class var using objects
TestClass object at 0x02091DD0: [1, 2, 3, 4,
'a', True]
TestClass object at 0x0218C710: [1, 2, 3, 4,
'a', True]
8 Display class var using class: [1, 2, 3, 4,
'a', True]
```

Both objects see the change that was made to the list even though that change was made independently of either object.

A word of caution is in order

This sort of behavior can lead to programming errors that are very easy to make and very difficult to find and fix. In my opinion, you should use mutable class variables very rarely if at all. On the other hand, class variables that are not mutable, such as tuples, can be very useful and are much safer to use. Even in that case, however, one immutable object can be replaced by a different immutable object so even the use of immutable objects is not totally safe.

Shadowing or hiding a class variable

If an instance variable belonging to an object has the same name as a class variable in the class from which the object was instantiated, the instance variable will "shadow" or hide the class variable insofar as that object is

concerned. This is illustrated by the program shown in [Listing 17](#). The output from the program is shown in [Figure 12](#).

Class definition with a shadowing method

The class definition in [Listing 11](#) contains a class variable named **classVar** . The class definition also contains a method named **shadowClassVariable** . If that method is executed on an object instantiated from the class, it will add a new instance variable to the object named **classVar** . From that point forward, the class variable named **classVar** won't be visible insofar as that object is concerned unless it accesses the class variable using the name of the class, **TestClass** .

Listing 11 . Class definition with a shadowing method.

Listing 11 . Class definition with a shadowing method.

```
class TestClass(object):
    classVar = 1234

    #Note, the following code adds a new
    instance variable named classVar
    # to the object, which "shadows" or hides
    the actual class variable
    # named classVar insofar as this object is
    concerned.
    def shadowClassVariable(self):
        self.classVar = "ABCD"

    def printClassVar(self):
        print(str(self)[10:40] + ": " +
              str(self.classVar))
```

[Listing 11](#) also defines a method named **printClassVar** . This method concatenates and prints a portion of the object identifier (*self*) with the contents of **classVar** (*as seen by the object*) .

Instantiate and display classVar in two objects

The code in [Listing 12](#) instantiates two different objects from the class named **TestClass** and saves the objects' references in the variables named **ref01** and **ref02** .

Listing 12 . Instantiate and display classVar in two objects.

```
print("1  Instantiate two objects")
ref01 = TestClass()
ref02 = TestClass()

print("2  Display classVar using objects")
ref01.printClassVar()
ref02.printClassVar()
```

The code in [Listing 12](#) also uses those references to call the method named **printClassVar** on each object. This causes the contents of **classVar** to be printed as seen by each object at this point in the program. The results are shown in [Figure 7](#).

As you have probably already figured out, each object gets and prints the contents of the class variable named **classVar** causing matching output values to be seen in [Figure 7](#).

Figure 7 . Output from the coded in Listing 12.

```
1 Instantiate two objects
2 Display classVar using objects
TestClass object at 0x02201DD0: 1234
TestClass object at 0x022FC710: 1234
```

Modify classVar in one object

The code in [Listing 13](#) calls the **shadowClassVariable** method on the object referred to by the variable named **ref01** . As I described [earlier](#), this call will add a new instance variable named **classVar** to the object. From that point forward, the class variable named **classVar** won't be visible insofar as that object is concerned unless it accesses the class variable using the name of the class, **TestClass** .

Listing 13 . Modify classVar in one object.

```
print("3  Shadow the classVar in one object")
ref01.shadowClassVariable()

print("4  Display classVar using objects")
ref01.printClassVar()
ref02.printClassVar()
```

Once again, The code in [Listing 13](#) also uses the objects' references to call the method named **printClassVar** on each object. This causes the contents of **classVar** to be printed as seen by each object at this point in the program. The results are shown in [Figure 8](#).

Figure 8 . Output from the code in Listing 13.

Figure 8 . Output from the code in Listing 13.

```
3 Shadow the classVar in one object
4 Display classVar using objects
TestClass object at 0x02201DD0: ABCD
TestClass object at 0x022FC710: 1234
```

The two output values in [Figure 8](#) no longer match. This is because the object referred to by **ref01** no longer sees the class variable named **classVar** . Instead, it sees the new instance variable named **classVar** , which contains a different value.

Modify the contents of the class variable

The code in [Listing 14](#) uses the name of the class, **TestClass** , to change the value stored in its class variable named **classVar** . Then the objects' reference are used to once again print the contents of **classVar** as seen by each object.

Listing 14 . Modify the contents of the class variable.

Listing 14 . Modify the contents of the class variable.

```
print("5  Modify classVar using class")
TestClass.classVar = True

print("6  Display classVar using objects")
ref01.printClassVar()
ref02.printClassVar()
```

The results are shown in [Figure 9](#). The change in the value of the class variable is reflected in the output produced by the object referred to by **ref02** . However, the object referred to by **ref01** is blind to that change because the class variable is shadowed or hidden by an instance variable having the same name in that object.

Figure 9 . Output from the code in Listing 14.

```
6 Display classVar using objects
TestClass object at 0x02201DD0: ABCD
TestClass object at 0x022FC710: True
```

Run the programs

I encourage you to copy the code from [Listing 15](#), [Listing 16](#), and [Listing 17](#). Execute the code and confirm that you get the same results as those

shown. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Visualize the programs

I also encourage you to create [visualizations](#) for the code in [Listing 15](#), [Listing 16](#), and [Listing 17](#). Step through the programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the use of class variables in Python.

Complete program listings

Complete listings of the programs discussed in this module, along with the outputs produced by those programs are provided below.

Listing 15 . Complete program listing.

Listing 15 . Complete program listing.

```
# This program demonstrates the modification
of a class after it has been
# used to instantiate objects.
#-----
-----
class TestClass(object):
    def addClassVar(self,data):
        TestClass.classVar02 = data

print("Instantiate and display two objects of
TestClass")
ref01 = TestClass()
print(ref01)
ref02 = TestClass()
print(ref02)

print("Add a class variable directly.")
TestClass.classVar01 = "ABCD"

print("Add a class variable via an object.")
ref01.addClassVar("1234")

print("Display both class variables via one
object")
print("ref01: " + ref01.classVar01)
print("ref01: " + ref01.classVar02)

print("Display both class variables via the
other object")
print("ref02: " + ref02.classVar01)
print("ref02: " + ref02.classVar02)
```


Figure 10 . Output from the code in Listing 15.

```
Instantiate and display two objects of
TestClass
<__main__.TestClass object at 0x02152930>
<__main__.TestClass object at 0x02152E70>
Add a class variable directly.
Add a class variable via an object.
Display both class variables via one object
ref01: ABCD
ref01: 1234
Display both class variables via the other
object
ref02: ABCD
ref02: 1234
```

Listing 16 . Complete program listing.

```
# This program demonstrates that class
variables are shared among the objects
# instantiated from a class and that changes
to class variables impact the
# objects instantiated from the class.
#-----
-----
class TestClass(object):
    classVar = [1,2,3,4]
```

Listing 16 . Complete program listing.

```
def modifyClassVar01(self):
    self.classVar.append("a")

def printClassVar01(self):
    print(str(self)[10:40] + ": " +
str(self.classVar))

print("1  Display class var using class: " +
str(TestClass.classVar))

print("2  Instantiate two objects")
ref01 = TestClass()
ref02 = TestClass()

print("3  Display class var using objects")
ref01.printClassVar01()
ref02.printClassVar01()

print("4  Modify class var using object")
ref01.modifyClassVar01()

print("5  Display class var using objects")
ref01.printClassVar01()
ref02.printClassVar01()

print("6  Modify class var using class")
TestClass.classVar.append(True)

print("7  Display class var using objects")
ref01.printClassVar01()
ref02.printClassVar01()

print("8  Display class var using class: " +
str(TestClass.classVar))
```

Listing 16 . Complete program listing.

Figure 11 . Output from the code in Listing 16.

```
1 Display class var using class: [1, 2, 3, 4]
2 Instantiate two objects
3 Display class var using objects
TestClass object at 0x02091DD0: [1, 2, 3, 4]
TestClass object at 0x0218C710: [1, 2, 3, 4]
4 Modify class var using object
5 Display class var using objects
TestClass object at 0x02091DD0: [1, 2, 3, 4,
'a']
TestClass object at 0x0218C710: [1, 2, 3, 4,
'a']
6 Modify class var using class
7 Display class var using objects
TestClass object at 0x02091DD0: [1, 2, 3, 4,
'a', True]
TestClass object at 0x0218C710: [1, 2, 3, 4,
'a', True]
8 Display class var using class: [1, 2, 3, 4,
'a', True]
```

Listing 17 . Complete program listing.

```
# This program illustrates that an instance
variable will shadow or hide a
# class variable with the same name.
#-----
-----
class TestClass(object):
    classVar = 1234

    #Note, the following code adds a new
instance variable named classVar
    # to the object, which "shadows" or hides
the actual class variable
    # named classVar insofar as this object is
concerned.
    def shadowClassVariable(self):
        self.classVar = "ABCD"

    def printClassVar(self):
        print(str(self)[10:40] + ": " +
str(self.classVar))

print("1  Instantiate two objects")
ref01 = TestClass()
ref02 = TestClass()

print("2  Display classVar using objects")
ref01.printClassVar()
ref02.printClassVar()

print("3  Shadow the classVar in one object")
ref01.shadowClassVariable()

print("4  Display classVar using objects")
```

Listing 17 . Complete program listing.

```
ref01.printClassVar()  
ref02.printClassVar()  
  
print("5  Modify classVar using class")  
TestClass.classVar = True  
  
print("6  Display classVar using objects")  
ref01.printClassVar()  
ref02.printClassVar()
```

Figure 12 . Output from the code in Listing 17.

```
1 Instantiate two objects  
2 Display classVar using objects  
TestClass object at 0x02201DD0: 1234  
TestClass object at 0x022FC710: 1234  
3 Shadow the classVar in one object  
4 Display classVar using objects  
TestClass object at 0x02201DD0: ABCD  
TestClass object at 0x022FC710: 1234  
5 Modify classVar using class  
6 Display classVar using objects  
TestClass object at 0x02201DD0: ABCD  
TestClass object at 0x022FC710: True
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1440-Class Variables
- File: Itse1359-1440.htm
- Published: 10/27/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1450-Inheritance

You will learn a little about Python class inheritance in this module.

Table of Contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Inheritance](#)
 - [Multiple inheritance](#)
 - [Very brief coverage](#)
- [Discussion and sample code](#)
 - [Define the superclass](#)
 - [Define the subclass](#)
 - [Exercise the classes](#)
 - [Display the value of the class variable](#)
 - [Instantiate a Subclass object](#)
 - [Modify and then print the value in superclassVar](#)
- [Run the program](#)
- [Visualize the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Earlier modules provided

- an overview of classes and objects in Python,
- an explanation of the **self** word in Python class definitions,
- a detailed explanation of **instance variables** in Python class definitions, and
- some of the details regarding **class variables** in Python class definitions.

What you will learn

You will learn a little about Python *class inheritance* in this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Output from the code in Listing 3.
- [Figure 2](#). Output from the code in Listing 4.
- [Figure 3](#). Output from the code in Listing 5.
- [Figure 4](#). Output from code in Listing 6.

Listings

- [Listing 1](#). Define the superclass.
- [Listing 2](#). Define a subclass that extends or inherits from Superclass.
- [Listing 3](#). Display the value of the class variable.
- [Listing 4](#). Instantiate a Subclass object.
- [Listing 5](#). Modify and then print the value in superclassVar.
- [Listing 6](#). Complete program listing.

General background information

I told you in an earlier module that OOP is an abbreviation for Object-Oriented Programming. I told you that most books on OOP will tell you that in order to understand OOP, you need to understand **the following three concepts** :

- Encapsulation
- Inheritance
- Polymorphism

I have discussed encapsulation in some detail in previous modules. As near as I can tell, unlike C++ and Java, Python does not support polymorphism, at least not in any significant way.

Inheritance

C++ and Java support two forms of polymorphism:

- Compile-time polymorphism

- Runtime polymorphism

Both of these depend on the "strongly-typed" nature of C++ and Java. Runtime polymorphism using virtual functions and base-class pointers also depends on inheritance.

Because Python is a "weakly-typed" (*if typed at all*) programming language, I don't know how to implement either form of polymorphism using Python. (*However, if I am wrong on this, I will be happy to learn how to implement polymorphism in Python.*)

Inheritance is extremely important in C++ and Java primarily because of the use of runtime polymorphism in program design. However, due to the lack of polymorphism, inheritance in Python is not nearly as important as in those other two languages. Although it may have some other uses, inheritance in Python appears mainly to be a convenient way to reduce the amount of typing required to write a program.

According to [tutorialspoint--Python Objected Oriented](#), Python inheritance is

"The transfer of the characteristics of a class to other classes that are derived from it."

While that can sometimes be a useful capability, it is simply one of many useful capabilities that exist in Python

Multiple inheritance

C++ supports a concept know as *multiple inheritance* , which often results in significant programming errors for novice programmers. Multiple inheritance is so problematic that the designers of Java excluded it from the language and replaced it with a safer alternative known as the *Java interface* .

Python also supports multiple inheritance. Because of the general lack of safety nets in Python, in my opinion, multiple inheritance is more problematic for novice programmers in Python than in C++. My advice -- don't use Python multiple inheritance for any serious programming effort until you become a very capable Python programmer.

Very brief coverage

As a result of the issues mentioned above, this module will provide only a very brief treatment of inheritance in Python. If you need to learn more about inheritance, I will refer you to the following documents that are available on the web as of October 2014.

- [tutorialspoint--Python Objected Oriented](#)
- [The Python Tutorial -- Inheritance](#)

Discussion and sample code

I will present and explain a simple program to give you a taste of inheritance. A complete listing of the program is provided in [Listing 6](#). The output produced by running the program is shown in [Figure 4](#). As usual, I will break the program down and explain it in fragments.

Define the superclass

[Listing 1](#) defines a class named **Superclass** that will be the *superclass* or the *base class* for another class named **Subclass**. In other words, the class named **Subclass** will *inherit from* the class named **Superclass** or will *extend* the class named **Superclass**, whichever jargon you prefer. (A [visualization of the program](#) uses the word *extends* to indicate inheritance.)

Depending on your preferred jargon, **Superclass** will be the *base class* or the *superclass*. **Subclass** will be the *derived class* or the *subclass*.

Listing 1 . Define the superclass.

```
#Define the superclass
class Superclass(object):
    superclassVar = True

    def __init__(self,name):
        Superclass.superclassVar = name
        self.friend = "Tom"
```

The code in [Listing 1](#) creates a class variable named **superclassVar** and initializes its value to **True** .

The code in [Listing 1](#) also defines an initializer or constructor method named **__init__** . When this method is executed, it changes the value stored in **superclassVar** from **True** to the value of an incoming parameter named **name** .

The **__init__** method also creates an instance variable named **friend** in the object referred to by **self** and initializes that variable to contain **"Tom"** .

Define the subclass

The class named **Subclass** , which extends **Superclass** , is defined in [Listing 2](#). This class also defines a constructor method named **__init__** , which in turn calls the **__init__** method of **Superclass** , passing its own incoming parameters to the superclass constructor method.

Listing 2 . Define a subclass that extends or inherits from Superclass.

```
#Define a subclass that extends or inherits
from Superclass
class Subclass(Superclass):
    def __init__(self,name):
        Superclass.__init__(self,name)
        print(self.friend)
```

The method named **__init__** in **Subclass** also prints the value of the variable named **friend** when it is executed.

When you instantiate a new object of a class that contains a method named **__init__**, that method is automatically called and a reference to the object is automatically passed as the first parameter to the method. However, if you write code that explicitly calls **__init__**, you must explicitly pass that parameter along with any other required parameters as shown in Listing 2.

Exercise the classes

The remaining program code is designed to exercise the classes and to illustrate the behavior of inheritance in Python. The code that exercises the classes begins in [Listing 3](#).

Display the value of the class variable

The code in [Listing 3](#) gets and prints the initial value stored in the class variable named **superclassVar** , producing the output shown in [Figure 1](#).

Listing 3 . Display the value of the class variable.

```
#Code that exercises the classes  
print(Superclass.superclassVar)
```

As you can see, at this point in the program, the class variable contains the value **True** that was placed there when the class variable was created by the code in [Listing 1](#).

Figure 1 . Output from the code in Listing 3.

True

Instantiate a Subclass object

The code in [Listing 4](#) instantiates an object of the class named **Subclass** passing "**Joe**" as a parameter to the **__init__** method belonging to that

class. (As you will recall, that `__init__` method is automatically called when the object is instantiated.)

[Figure 2](#) shows the output produced by executing the code in [Listing 4](#).

Listing 4 . Instantiate a Subclass object.

```
ref = Subclass("Joe")  
  
print(ref.superclassVar)
```

As you saw in [Listing 2](#), the `__init__` method in the class named **Subclass** calls the method named `__init__` in the class named **Superclass** passing both of its incoming parameters as parameters to the `__init__` method in the class named **Superclass** . As you saw in [Listing 1](#), that method replaces the value **True** in **superclassVar** with one of its incoming parameters, which in this case is **"Joe"** .

The code in the method named `__init__` in the class named **Subclass** in [Listing 2](#) also prints the value of the variable named **friend** (which is created and initialized by the `__init__` method in the class named **Superclass** in [Listing 1](#)) . This produces the first line of output text (Tom) shown in [Figure 2](#).

The code in [Listing 4](#) also uses the reference to the **Subclass** object to get and print the value of the class variable named **superclassVar** from the class named **Superclass** , producing the second line of output text in [Figure 2](#) (Joe) .

Figure 2 . Output from the code in Listing 4.

Tom
Joe

Modify and then print the value in superclassVar

Finally, the code in [Listing 5](#) uses the reference to the **Subclass** object to modify and then print the value stored in the class variable of the **Superclass** class named **superclassVar** .

Listing 5 . Modify and then print the value in superclassVar.

```
ref.superclassVar = "Sue"  
print(ref.superclassVar)
```

[Figure 3](#) shows the output produced by executing the code in [Listing 5](#).

Figure 3 . Output from the code in Listing 5.

Figure 3 . Output from the code in Listing 5.

Sue

Run the program

I encourage you to copy the code from [Listing 6](#). Execute the code and confirm that you get the same results as those shown in [Figure 4](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Visualize the program

I also encourage you to create a [visualization](#) for the code in [Listing 6](#). Step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand inheritance in Python.

Complete program listing

A complete listing of the program discussed in this module is provided in [Listing 6](#) below. The output produced by the code in [Listing 6](#) is shown in [Figure 4](#).

Listing 6 . Complete program listing.

Listing 6 . Complete program listing.

```
# Illustrates inheritance
#-----
-----
#Define the superclass
class Superclass(object):
    superclassVar = True

    def __init__(self,name):
        Superclass.superclassVar = name
        self.friend = "Tom"

#Define a subclass that extends or inherits
from Superclass
class Subclass(Superclass):
    def __init__(self,name):
        Superclass.__init__(self,name)
        print(self.friend)

#Code that exercises the classes
print(Superclass.superclassVar)
ref = Subclass("Joe")
print(ref.superclassVar)
ref.superclassVar = "Sue"
print(ref.superclassVar)
```

Figure 4 . Output from code in Listing 6.

Figure 4 . Output from code in Listing 6.

```
True
Tom
Joe
Sue
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1450-Inheritance
- File: Itse1359-1450.htm
- Published: 10/27/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales

nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1450r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1450-Inheritance.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1450-Inheritance* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
class Superclass(object):
    superclassVar = True

    def __init__(self):
        self.superclassVar = "Tom"

class Subclass(Superclass):
    def __init__(self, name):
        self.brother = name

print(Superclass.superclassVar)
ref = Subclass("Joe")
print(ref.superclassVar)
print(ref.brother)
```

Figure 2 . Question 1 possible output.

True
Tom
Joe

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

False. The actual output is shown in [Figure 3](#). Unlike Java and C++, when you instantiate an object of a subclass, the subclass constructor does not automatically call the constructor for the superclass. If you want that to happen, you must explicitly write code in the subclass constructor to make it happen.

Figure 3 . Question 1 actual output.

Figure 3 . Question 1 actual output.

True
True
Joe

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1450r-Review
- File: Itse1359-1450r.htm
- Published: 10/27/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com

showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1460-Turtle Graphics

This module introduces the student to Python's Turtle graphics.

Table of contents

- [Preface](#)
- [Discussion](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming using Python's [Turtle graphics](#) module.

Discussion

Now that you know something about classes and objects, it's time to gain some insight into what can be done with them. I know of no better way to gain that insight than through the use of Python's [Turtle graphics](#). Rather than trying to explain Turtle graphics myself, I am going to refer you to several online resources that explain the topic much better than I am equipped to do.

First, I recommend that you begin with the [Hello Little Turtles!](#) chapter of the online interactive book titled [How to Think Like a Computer Scientist - Learning with Python: Interactive Edition 2.0](#). Be sure to watch the [video](#). To get a quick idea of the capability offered by turtles, go to [this page](#) and press the **Run** button below the image of the spirals.

In parallel with that, I recommend that you also study the non-interactive version of the chapter titled [Hello, little turtles!](#) in the non-interactive version of the book titled [How to Think Like a Computer Scientist: Learning with Python 3](#). Each version has advantages and disadvantages over the other.

If you Google "python turtle graphics", you will likely find dozens of resources. Here are a few of them:

- [Hello Little Turtles!](#) - interactive Ebook
- [Hello, little turtles!](#) - non-interactive Ebook
- [Python documentation, 24.1. turtle - Turtle graphics](#)
- [Background: Drawing Graphics](#)
- [Notes on using Python's turtle built-in commands](#)
- [Python for beginners](#)
- [turtle - Turtle graphics for Tk](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1460-Turtle Graphics
- File: Itse1359-1460.htm
- Published: 03/15/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a

book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1510-Reading and Writing Text Files

You will learn how to read, write, and append to text files using Python in this module.

Table of Contents

- [Preface](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Open a text file for writing](#)
 - [Write text and close the file](#)
 - [Read and print the file contents with a for loop](#)
 - [Append more text to the file](#)
 - [Read and print the file contents with a while loop](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you will learn

You will learn how to read, write, and append to text files using Python in this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Output from the code in Listing 3.
- [Figure 2](#). Output from the code in Listing 5.
- [Figure 3](#). Output produced by the code in Listing 6.

Listings

- [Listing 1](#). Open a text file for writing.
- [Listing 2](#). Write text and close the file.
- [Listing 3](#). Read and print the file contents with a for loop.
- [Listing 4](#). Append more text to the file.
- [Listing 5](#). Read and print the file contents with a while loop.
- [Listing 6](#). Complete program listing.

General background information

From time to time you may need to read, write, or append to files. I will present and explain a relatively simple program in this module that will get you started down that path. However, this module is not intended to be a comprehensive tutorial on file i/o. Instead, I will refer you to other online resources for more information if and when you need it. The following web sites provide important information in this regard:

- [tutorialspoint -- Python Files I/O](#)
- [The Python Tutorial -- Reading and Writing Files](#)
- [Python Library Reference \(version 2.3\)](#)

Discussion and sample code

The program that I will present and explain in this module begins by writing a text file named "**SampleTextFile.txt**" into the current directory and populating it with five lines of text. If a file already exist having that name in the current directory, it will be overwritten by the new file.

Then the program uses a **for** loop as an iterator to read and print each line of text from the file.

After that, the program appends two additional lines of text to the file and uses a **while** loop to read and print the seven lines of text from the file.

A complete listing of the program is provided in [Listing 6](#). The output from the program is shown in [Figure 3](#). As usual, I will break the program down and explain it in fragments. The first fragment is shown in [Listing 1](#).

Open a text file for writing

[Listing 1](#) calls the built-in function named [open](#) to create and return a **file** object in *write mode* . The reference to the file object is stored in the variable named **theFile** . The *write mode* is indicated by the "w" as the second parameter to the **open** function. The description of the [open](#) function at [The Python Standard Library -- 2. Built-in Functions](#) describes eight

different modes that can be specified when a **file** object is created. The program in this module will use the following three modes:

- 'w' -- open for writing, truncating the file first
- 'r' -- open for reading (default)
- 'a' -- open for writing, appending to the end of the file if it exists

Listing 1 . Open a text file for writing.

```
#Open a file object for writing.  
theFile = open("SampleTextFile.txt", "w")
```

Write text and close the file

[Listing 2](#) calls the [write](#) method five times in succession to write five strings into the file.

Listing 2 . Write text and close the file.

Listing 2 . Write text and close the file.

```
#Write several lines of text to the file. Note
the requirement to explicitly
#provide the newline at the end of each line
of text.
theFile.write("This is a sample text file.\n")
theFile.write("Second line of text.\n")
theFile.write("Third line.\n")
theFile.write("Fourth line.\n")
theFile.write("Last line.\n")

#Be sure to close the file
theFile.close()
```

Once you have a **file** object, there are a variety of methods that you can call on that object, one of which is the method named **write** . The best description that I could find for that set of methods is located at [File Objects](#) , which is part of the documentation for Python version 2.3. (*Hopefully it is still correct for Python version 3.*)

That document provides the following description for the **write** method:

Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the `flush()` or `close()` method is called.

[Listing 2](#) calls the **close** method on the file after the strings are written to the file to ensure that the data is actually moved from the output buffer into the file. The description of the **close** method at [File Object](#) reads as follows:

Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a `ValueError` after the file has been closed. Calling `close()` more than once is allowed.

Read and print the file contents with a for loop

[Listing 3](#) shows one way to read the contents of a text file, line-by-line, using a **for** loop as an iterator. Note that the "r" mode is the default but it is shown here for clarity.

Listing 3 . Read and print the file contents with a for loop.

```
#Open the file object in read mode.
theFile = open("SampleTextFile.txt","r")

#Read and print the lines of text.
for line in theFile:
    print(line,end='')

#Close the file
theFile.close()
```

The output produced by the code in [Listing 3](#) is shown in [Figure 1](#). As you can see, the output matches the text written into the file in [Listing 2](#).

Figure 1 . Output from the code in Listing 3.

```
This is a sample text file.  
Second line of text.  
Third line.  
Fourth line.  
Last line.
```

The syntax of the call to the **print** function with two parameters in [Listing 3](#) is probably different from what you have seen in my previous modules. By default, the **print** function terminates each output line with a *newline* . Because the lines of text in the file already contain a newline, without the second parameter shown in [Listing 3](#), the output shown in [Figure 1](#) would be double-spaced. The inclusion of the second parameter in the call to the **print** function in [Listing 3](#) prevents the **print** function from inserting an extra newline and inserts an empty string instead.

Append more text to the file

The code in [Listing 4](#) begins by printing a blank line to cause the output shown in [Figure 3](#) to be more readable.

Then the code in [Listing 4](#)

- re-opens the file for appending,
- writes two additional text lines into the file, and
- closes the file.

Listing 4 . Append more text to the file.

```
print()#blank line to separate the output text

#Append some text to the file.
#Open the file object for appending.
theFile = open("SampleTextFile.txt","a")

#Append two more lines of text to the file.
theFile.write("First appended line.\n")
theFile.write("Second appended line.\n")

#Close the file
theFile.close()
```

Read and print the file contents with a while loop

The **readline** method of a **file** object will read and return the next line from the file. An empty string is returned when the *end of file* is encountered. *(There are some subtle ramifications that you can learn about at [File Objects](#).)*

The code in [Listing 5](#) uses the **readline** method in a **while** loop to read and print each line of text in the file.

Listing 5 . Read and print the file contents with a while loop.

Listing 5 . Read and print the file contents with a while loop.

```
#Open the file object in default read mode.
theFile = open("SampleTextFile.txt")

#Read and print the lines of text using a
different approach.
line = theFile.readline()
print(line,end='')
while line != "":
    line = theFile.readline()
    print(line,end='')

#Close the file
theFile.close()
```

[Figure 2](#) shows the output produced by the code in [Listing 5](#).

Figure 2 . Output from the code in Listing 5.

Figure 2 . Output from the code in Listing 5.

```
This is a sample text file.  
Second line of text.  
Third line.  
Fourth line.  
Last line.  
First appended line.  
Second appended line.
```

If you compare [Figure 2](#) with [Figure 1](#), you will see that the two lines of text that were appended by the code in [Listing 4](#) are shown in [Figure 2](#).

Run the program

I encourage you to copy the code from [Listing 6](#). Execute the code and confirm that you get the same results as those shown in [Figure 3](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program discussed in this module is provided in [Listing 6](#). The output produced by this program is shown in [Figure 3](#).

Listing 6 . Complete program listing.

```
# Illustrates how to write, read, and append to a  
# text file.  
#-----  
-----
```

```
#Open a file object for writing.
theFile = open("SampleTextFile.txt","w")

#Write several lines of text to the file. Note the
requirement to explicitly
#provide the newline at the end of each line of
text.
theFile.write("This is a sample text file.\n")
theFile.write("Second line of text.\n")
theFile.write("Third line.\n")
theFile.write("Fourth line.\n")
theFile.write("Last line.\n")
#Be sure to close the file
theFile.close()

#Open the file object in read mode. Note that the
"r" mode is the default
#but it is shown here for clarity.
theFile = open("SampleTextFile.txt","r")

#Read and print the lines of text.
for line in theFile:
    print(line,end='')

#Close the file
theFile.close()

print()#blank line to separate the output text
#Append some text to the file.
#Open the file object for appending.
theFile = open("SampleTextFile.txt","a")

#Append two more lines of text to the file.
theFile.write("First appended line.\n")
theFile.write("Second appended line.\n")
```



```
#Close the file
theFile.close()

#Open the file object in default read mode.
theFile = open("SampleTextFile.txt")

#Read and print the lines of text using a
different approach.
line = theFile.readline()
print(line,end='')
while line != "":
    line = theFile.readline()
    print(line,end='')

#Close the file
theFile.close()
```

Figure 3 . Output produced by the code in Listing 6.

Figure 3 . Output produced by the code in Listing 6.

```
This is a sample text file.  
Second line of text.  
Third line.  
Fourth line.  
Last line.
```

```
This is a sample text file.  
Second line of text.  
Third line.  
Fourth line.  
Last line.  
First appended line.
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1510-Reading and Writing Text Files
- File: Itse1359-1510.htm
- Published: 10/28/14
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1510r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1510-Reading and Writing Text Files.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1510-Reading and Writing Text Files* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) produces the output shown in [Figure 2](#).

Figure 1 . Question 1 program code.

```
data = ["A", "S", "D", "F", "G", "H"]

theFile = open("SampleTextFile.txt", "w")
theFile.write("First line.\n")
theFile.close()

theFile = open("SampleTextFile.txt", "a")
for index in range(1,6,2):
    theFile.write(data[index])

theFile.write("Last line.\n")

theFile = open("SampleTextFile.txt", "r")
for line in theFile:
    print(line, end=' ')

theFile.close()
```

Figure 2 . Question 1 possible output.

```
First line.  
S  
F  
H  
Last line.
```

Go to [answer 1](#)

Figure index

- [Figure 1](#). Question 1 program code.
- [Figure 2](#). Question 1 possible output.
- [Figure 3](#). Question 1 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 1

False. The actual output is shown in [Figure 3](#). The **write** method does not automatically append a newline character onto the end of the text being

written. If you want it there, you must explicitly put it there as shown by the code for the first and last lines in [Figure 1](#).

Figure 3 . Question 1 actual output.

```
First line.  
SFHLast line.
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1510r-Review
- File: Itse1359-1510r.htm
- Published: 10/28/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1610-GUI Programming

This module explains the ins and outs of creating a simple GUI using Python and tkinter.

Table of Contents

- [Preface](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Tk and tkinter](#)
 - [What is a GUI?](#)
 - [What is TCL?](#)
 - [What is TK?](#)
 - [What is tkinter?](#)
 - [Not the only GUI alternative](#)
 - [A big and complex topic](#)
- [Discussion and sample code](#)
 - [A sample program](#)
 - [Will discuss in fragments](#)
 - [The root](#)
 - [Import statements](#)
 - [The calculate function](#)
 - [Add a Frame widget](#)
 - [The grid geometry manager](#)
 - [Columns and rows](#)
 - [A sticky situation](#)

- [StringVar](#)
 - [The Entry widget](#)
 - [The width argument](#)
 - [The textvariable argument](#)
 - [Make the Entry field visible](#)
 - [A sticky demonstration](#)
 - [A Label and a Button](#)
 - [Three more Label widgets](#)
 - [Finishing touches](#)
 - [Now back to the calculate function](#)
 - [The input value in feet](#)
 - [Either save the float or pass](#)
 - [Calculate and display the result in meters](#)
 - [Make everything run](#)
- [Run the program](#)
 - [Complete program listing](#)
 - [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module is not intended to be a stand-alone tutorial. Instead, it is intended to be an annotated guide to other freely available online resources on the topic.

What you will learn

You will learn how to create a simple GUI using Python and **tkinter** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: the Figures in this module are bitmap images and are probably not accessible using an audio screen reader or a braille display. The Listings are text and should be accessible. Therefore, this module is only partially accessible to blind students.

Because the module is not accessible to blind students, material from this module will not be used for assignments and will not be included on tests. However, sighted students are strongly encouraged to study the material for their own education. Blind and visually impaired students are also encouraged to learn as much from the module as they can.)

Figures

- [Figure 1](#). Image of a tkinter test from the command line.
- [Figure 2](#). Output from Feet to Meters converter program.
- [Figure 3](#). Output from Listing 1.
- [Figure 4](#). A sticky demonstration.

Listings

- [Listing 1](#). The root.
- [Listing 2](#). Add a Frame widget.
- [Listing 3](#). StringVar objects.
- [Listing 4](#). The Entry widget.
- [Listing 5](#). A Label and a Button.
- [Listing 6](#). Three more Label widgets.
- [Listing 7](#). Finishing touches.
- [Listing 8](#). The calculate function.
- [Listing 9](#). Complete program listing.

General background information

Tk and tkinter

What is a GUI ?

According to [Wikipedia](#),

"In computing, a graphical user interface (GUI... sometimes pronounced "gooey" ...) is a type of interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces ... which require commands to be typed on the keyboard."

You probably interface with most of the programs that you use on a daily basis through a GUI. If you use Python to create programs that will be used by non programmers, you may need to provide GUIs for those programs.

What is TCL?

According to the [Tcl Developer Xchange](#),

"Tcl (Tool Command Language) is a very powerful but easy to learn dynamic programming language, suitable for a very wide range of uses, including web and desktop applications, networking, administration, testing and many more. Open source and business-friendly, Tcl is a mature yet evolving language that is truly cross platform, easily deployed and highly extensible."

What is TK?

Also according to the [Tcl Developer Xchange](#),

"Tk is a graphical user interface toolkit that takes developing desktop applications to a higher level than conventional approaches. Tk is the standard GUI not only for Tcl, but for many other dynamic languages, and can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more."

What is tkinter?

According to [The Python Standard Library -- 25. Graphical User Interfaces with Tk](#),

"Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the tkinter package, and its extension, the tkinter.tix and the tkinter.ttk modules. ... To use tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. tkinter...usually comes bundled with Python."

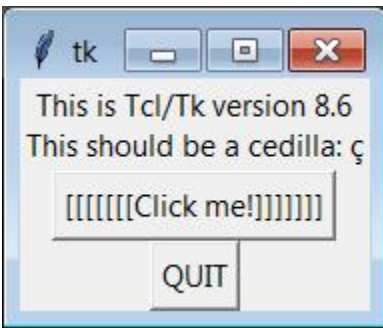
According to [The Python Standard Library -- 25.1 tkinter -- Python interface to Tcl/Tk](#),

"The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available

on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.) You can check that tkinter is properly installed on your system by running `python -m tkinter` from the command line; this should open a window demonstrating a simple Tk interface."

[Figure 1](#) shows the result of entering the command given above on the command line on my computer and then clicking the button labeled **Click me** several times. Clicking the QUIT button will cause the GUI to disappear from the screen as expected.

Figure 1 . Image of a tkinter test from the command line.



Not the only GUI alternative

You will find other GUI alternatives for Python listed and described at [The Python Standard Library -- 25.6. Other Graphical User Interface Packages](#). However, this module will concentrate on **tkinter**.

A big and complex topic

The design and implementation of graphical user interfaces is a huge topic that could fill an E-book in its own right. In this module, I will barely scratch the surface. In particular, I will present and explain one GUI program. Beyond that, I will refer you to several good online resources

where you can learn more about GUI programming in Python if you are interested.

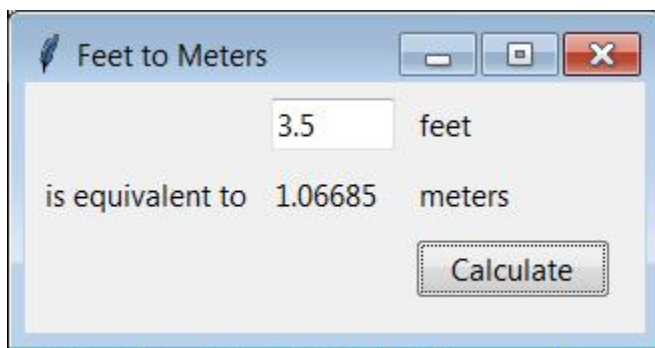
Discussion and sample code

A sample program

[Listing 9](#) presents an unmodified version of a program that was published by [Mark Roseman](#) at [TkDocs](#) under a [Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Canada \(CC BY-NC-SA 2.5 CA\)](#) license. This program GUI allows the user to enter a length value in feet. When the user presses a button labeled **Calculate**, the corresponding length value in meters is displayed. This process can be repeated until the user presses the X-button in the upper-right corner of the GUI, which terminates the program and removes the GUI from the screen.

[Figure 2](#) shows the output produced by running this program.

Figure 2 . Output from Feet to Meters converter program.



Will discuss in fragments

As is my custom, I will break the program down and discuss it in fragments. I will attempt to explain why the various parts of the program behave as they do. The first fragment is shown in [Listing 1](#).

The root

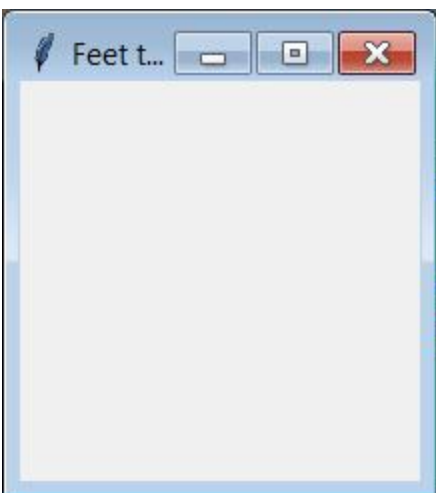
[Listing 1](#) shows a skeleton of the program with most of the code removed to expose the **root** and the *main loop* .

Listing 1 . The root.

```
from tkinter import *  
  
root = Tk()  
root.title("Feet to Meters")  
  
root.mainloop()
```

[Figure 3](#) shows the output produced by running the skeleton code in [Listing 1](#).

Figure 3 . Output from Listing 1.



According to [Hello, Tkinter](#),

"To initialize Tkinter, we have to create a Tk root widget. This is an ordinary window, with a title bar and other decoration provided by your window manager. You should only create one root widget for each program, and it must be created before any other widgets."

*(Note that according to [TkDocs](#), the name of the module was changed from **Tkinter** to **tkinter** with the release of Python 3.0.)*

The empty window that you see in [Figure 3](#) behaves like a standard Windows window on my machine. The three buttons in the upper-right corner behave as expected while the menu icon in the upper-left corner behaves as expected. *(The icon in the upper-left corner appears to be the TCL feather logo.)*

Import statements

[TkDocs](#) has this to say about the two import statements at the top of [Listing 9](#),

"These two lines tell Python that our program needs two modules. The first, "tkinter", is the standard binding to Tk, which when loaded also causes the existing Tk library on your system to be loaded. The second, "ttk", is Python's binding to the newer "themed widgets" that were added to Tk in 8.5."

That document goes on to provide a very interesting TIP regarding the use of the asterisk (*) with the first import statement and the non-use of the

asterisk with the second import statement. I will leave it as an exercise for the student to examine that material.

The calculate function

I'm going to skip over the **calculate** function (see [Listing 9](#)) at this point and return to discuss it in context later.

Add a Frame widget

The first statement in [Listing 2](#) creates a **Frame** widget and places it in the **root** .

Listing 2 . Add a Frame widget.

```
mainframe = ttk.Frame(root, padding="3 3 12 12")
mainframe.grid(column=0, row=0, sticky=(N, W, E, S))

mainframe.columnconfigure(0, weight=1)
mainframe.rowconfigure(0, weight=1)
```

According to TkDocs ,

"Next, we create a frame widget, which will hold all the content of our user interface, and place that in our main window. The "columnconfigure"/"rowconfigure" bits just tell Tk that if the main window is resized, the frame should expand to take up the extra space."

According to [Tkinter 8.5 reference: a GUI for Python](#),

"Like the Tkinter Frame widget, the ttk.Frame widget is a rectangular container for other widgets."

That document goes on to explain that the first argument to **Frame** is the parent of the **Frame** . In this case, the frame will be a child of **root** . After that, you can enter any of about nine different options, one of which is **padding** .

According to [Padding](#) at TkDocs,

"Normally, each column or row will be directly adjacent to the next, so that widgets will be right next to each other. This is sometimes what you want (think of a listbox and its scrollbar), but often you want some space between widgets. In Tk, this is called padding, and there are several ways you can choose to add it."

One of the ways to achieve the desired space between widgets is by using the syntax shown in the **padding** argument of [Listing 2](#) . This syntax inserts spaces (*in units of pixels*) on the four sides of the frame in the order west, north, east, and south or left, top, right, and bottom. (*Note that the integers in [Listing 2](#) are not separated by commas.*)

The grid geometry manager

Once you have a **Frame** widget, you can call various methods on it. One of those methods is **grid** as shown by the second line of code in [Listing 2](#).

According to [The Grid Geometry Manager](#) at TkDocs,

"Grid is one of several geometry managers available in Tk, but it's mix of power, flexibility and ease of use, along with its natural fit with today's layouts (that rely on alignment of widgets) make it the best choice for general use. There are other geometry managers: "pack" is also quite powerful, but harder to use and understand; "place" gives you complete control of positioning each element; we'll see even widgets like paned windows, notebooks, canvas and text can act as geometry managers."

Columns and rows

With the **Grid** geometry manager, a container of widgets such as the **root** or the **Frame** is subdivided into a grid of cells. The locations of widgets in the container are specified by column and row numbers.

According to [Columns and Rows](#) at TkDocs,

"Column and row numbers must be integers, with the first column and row starting at 0. You can leave gaps in column and row numbers (e.g. column 0, 1, 2, 10, 11, 12, 20, 21), which is handy if you plan to add more widgets in the middle of the user interface at a later time. The width of each column (or height of each row) depends on the width or height of the widgets contained within the column or row. This means when sketching out your user interface, and dividing it into rows and columns, you don't need to worry about each column or row being equal width."

The second statement in [Listing 2](#) shows that the **Frame** is placed in column 0 and row 0 (*the upper-left corner*) of the **root** .

A sticky situation

The sizes of the cells in a grid are determined by the sizes of the widgets placed in them. Therefore, the width of the cells in a column will be at least as wide as the widest widget in the column and the height of the cells in a row will be at least as high as the tallest widget placed in the row.

By default, if a widget is smaller than the cell in which it is placed, it will be placed in the upper-left corner of the cell. The **sticky** argument can be used to specify something other than the default. The value of the sticky argument consists of the compass directions N, S, E, and W as shown in the second statement of [Listing 2](#). (*Note that they are separated by commas.*)

Many different combinations of N, S, E, and W can be used to control the placement of the widget in the cell. A good explanation is provided in [Layout within the Cell](#) at TkDocs. The combination used in the second statement of [Listing 2](#) causes the **Frame** to be stretched in all four directions so as to be stuck to all four sides of the **root** . In other words, the **Frame** completely fills the available space within the **root** container.

The last two statements in [Listing 2](#) were explained by [According to TkDocs](#) above.

StringVar

To make a long story short, the **StringVar** objects shown in [Listing 3](#) are used to connect the string values for **feet** and **meters** to the **textvariable** arguments of an **Entry** widget and a **Label** widget, which you will see later.

(If you connect a widget to a **StringVar** object using a **textvariable** argument, a change in the contents of either will cause a corresponding change in the other. This is discussed in some detail at [25.1.6.4. Coupling Widget Variables](#).)

Listing 3 . StringVar objects.

```
feet = StringVar()  
meters = StringVar()
```

The Entry widget

In the [initial design stages](#) of the GUI, the author indicates that the widgets will be placed in three columns and three rows.

The first statement in [Listing 4](#) creates a new widget of the class [Entry](#) and saves the object's reference in the variable named **feet_entry** . According to [The Tkinter Entry Widget](#),

"The Entry widget is a standard Tkinter widget used to enter or display a single line of text."

The first argument is the parent -- in this case **mainframe** . This is followed by many possible optional arguments, one of which is **width** .

Listing 4 . The Entry widget.

```
feet_entry = ttk.Entry(mainframe, width=7,  
textvariable=feet)  
  
feet_entry.grid(column=2, row=1, sticky=(W,  
E))
```

The width argument

[The Tkinter Entry Widget](#) describes the **width** argument as follows:

"Width of the entry field, in character units. Note that this controls the size on screen; it does not limit the number of characters that can be typed into the entry field. The default width is 20 character."

The first statement in [Listing 4](#) sets the width to seven characters.

The textvariable argument

[The Tkinter Entry Widget](#) describes the **textvariable** argument as follows:

"Associates a Tkinter variable (usually a StringVar) to the contents of the entry field."

This argument is set equal to **feet** from [Listing 3](#). From this point forward, values entered into the data entry field shown in [Figure 2](#) will be saved in the **StringVar** named **feet**. This makes the value available to the **calculate** function that will be discussed later.

Make the Entry field visible

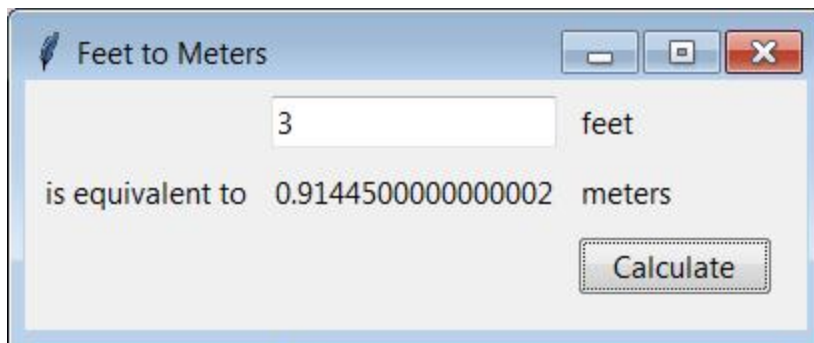
The author of the [program](#) explains that the first statement in [Listing 4](#) creates the **Entry** widget but does not make it visible in the GUI. In order to make it visible, you must tell Tk where to place it relative to other widgets. This is accomplished by the call to **grid** in the second statement in [Listing 4](#), which places the **Entry** widget in column 2 and row 1. *(Note that the author seems to have skipped column 0 and row 0 as described [earlier](#).)*

The **sticky** parameter in [Listing 4](#) instructs Tk to stretch the **Entry** widget horizontally and make it touch both sides of the cell. *(Note that manually stretching the size of the **root** on the screen does not stretch the size of the **Frame** or the widgets contained in the **Frame**.)*

A sticky demonstration

As an aside, I am going to demonstrate what happens with the sticky parameters in [Listing 4](#) when the column width changes. Compare [Figure 4](#) with [Figure 2](#)

Figure 4 . A sticky demonstration.



As you will see shortly, the numerical result of calculating the conversion from feet to meters is displayed in column 2 in the row immediately below the **Entry** widget. When that result requires more width than is available in the column by default, the width of the column is automatically increased. Because the right and left ends of the **Entry** widget stick to the sides of its cell in that column, the width of the **Entry** widget increases accordingly.

A Label and a Button

By now you should be catching on to the general scheme of things and less detailed instructions should be needed.

[Listing 5](#) creates a **Label** widget and a **Button** widget and places them in the rows and columns indicated by the **sticky** parameters.

Listing 5 . A Label and a Button.

```
ttk.Label(mainframe,  
textvariable=meters).grid(column=2, row=2,  
sticky=(W, E))  
ttk.Button(mainframe, text="Calculate",  
command=calculate).grid(  
    column=3, row=3, sticky=W)
```

A few things are different here:

- References to the widgets are not saved in ordinary Python variables because there will be no need to refer to them later in program code.

- The dot operator is used to call **grid** immediately upon the creation of each widget as opposed to calling **grid** in a separate statement as was done in [Listing 4](#).
- The **Button** widget uses a **text** argument to place the string "**Calculate**" on the face of the button.
- No text is placed in the **Label** . The empty **Label** will be used to display the results of converting feet to meters.
- The left end of the **Button** widget is stuck to the west side of the cell in which it resides. It is not stretched to the width of the cell as is the case with the **Entry** and **Label** widgets
- Last and perhaps most important, the **Button** widget uses the **command** argument to specify that the function named **calculate** will be executed whenever the user presses the button.

Three more Label widgets

[Listing 6](#) creates three more **Label** widgets and places them in the rows and columns shown.

Listing 6 . Three more Label widgets.

```
ttk.Label(mainframe,  
text="feet").grid(column=3, row=1, sticky=W)  
ttk.Label(mainframe, text="is equivalent  
to").grid(column=1, row=2, sticky=E)  
ttk.Label(mainframe,  
text="meters").grid(column=3, row=2, sticky=W)
```

There is nothing new in [Listing 6](#) so no explanation of that code should be required.

Finishing touches

The [author](#) of the program describes the first line of code in [Listing 7](#) as follows:

"The first line walks through all of the widgets that are children of our content frame, and adds a little bit of padding around each, so they aren't so scrunched together. We could have added these options to each "grid" call when we first put the widgets onscreen, but this is a nice shortcut."

Listing 7 . Finishing touches.

```
for child in mainframe.winfo_children():
    child.grid_configure(padx=5, pady=5)

feet_entry.focus()

root.bind('<Return>', calculate)
```

The [author](#) describes the second line of code in [Listing 7](#) as follows:

"The second line tells Tk to put the focus on our entry widget. That way the cursor will start in that field, so the user doesn't have to click in it before starting to type."

And finally , the [author](#) describes the third line of code in [Listing 7](#) as follows:

"The third line tells Tk that if the user presses the Return key (Enter on Windows) anywhere within the root window, that it should call our calculate routine, the same as if the user pressed the Calculate button."

Now back to the calculate function

The calculate function, which is shown in [Listing 8](#), is executed whenever the user presses the **calculate** button or hits the *Enter* key on the keyboard as described [above](#).

Listing 8 . The calculate function.

Listing 8 . The calculate function.

```
def calculate(*args):  
    try:  
        value = float(feet.get())  
        meters.set((0.3048 * value * 10000.0 +  
0.5)/10000.0)  
    except ValueError:  
        pass
```

The input value in feet

Any value that is entered into the **Entry** widget is stored in the **StringVar** object referred to by **feet** as shown in [Listing 3](#) and [Listing 4](#) and as described [above](#) .

The **StringVar** object has a method named **get** that returns the string stored in the object. The code in the **try** block of [Listing 8](#) calls the **get** method on the **StringVar** object and attempts to convert the string to type **float** .

Either save the float or pass

If the string cannot be converted to a **float** , a **ValueError** is thrown and control is passed to the **pass** statement. This causes the function to exit without doing anything.

If the string can be converted to a **float** , that float value is stored in the variable named **value** .

Calculate and display the result in meters

A **StringVar** object also has a method named **set** that can be used to store a value in the object. The code in the **try** block performs the arithmetic to convert the value in feet to a value in meters and then calls the **set** method on **meters** to store that value in the **StringVar** object referred to by **meters** in [Listing 3](#).

Storing a value in that **StringVar** object causes the value to appear in the **Label** immediately to the left of the word **meters** in [Figure 2](#) due to the **textvariable** argument in [Listing 5](#). Once again, if you connect a widget to a **StringVar** object using a **textvariable** argument, a change in the contents of either will cause a corresponding change in the other.

Make everything run

Finally, the last statement in the program, which is shown in [Listing 9](#), causes Tk to enter its *event-handling loop* and causes everything to run. The code in this loop will monitor for an event indicating that the user has pressed the **calculate** button. When that happens, the **calculate** function will be executed causing the conversion from feet to meters to take place and causing the results of the calculation to be displayed as shown in [Figure 2](#). This is an infinite loop that will continue to run and monitor for events until the program is terminated.

Run the program

I encourage you to copy the code from [Listing 9](#). Execute the code and confirm that you get results similar to those shown in [Figure 2](#) or [Figure 4](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is shown in [Listing 9](#).

Listing 9 . Complete program listing.

```
# Illustrates a Python GUI using tkinter.
# Published by Mark Roseman at TkDocs under a
Creative Commons
# Attribution-NonCommercial-ShareAlike 2.5
Canada (CC BY-NC-SA 2.5 CA) license.
#-----
-----
from tkinter import *
from tkinter import ttk

def calculate(*args):
    try:
        value = float(feet.get())
        meters.set((0.3048 * value * 10000.0 +
0.5)/10000.0)
    except ValueError:
        pass

root = Tk()
root.title("Feet to Meters")

mainframe = ttk.Frame(root, padding="3 3 12
12")
mainframe.grid(column=0, row=0, sticky=(N, W,
E, S))
mainframe.columnconfigure(0, weight=1)
mainframe.rowconfigure(0, weight=1)

feet = StringVar()
meters = StringVar()

feet_entry = ttk.Entry(mainframe, width=7,
textvariable=feet)
```


Listing 9 . Complete program listing.

```
feet_entry.grid(column=2, row=1, sticky=(W,
E))

ttk.Label(mainframe,
textvariable=meters).grid(column=2, row=2,
sticky=(W, E))
ttk.Button(mainframe, text="Calculate",
command=calculate).grid(
    column=3, row=3, sticky=W)

ttk.Label(mainframe,
text="feet").grid(column=3, row=1, sticky=W)
ttk.Label(mainframe, text="is equivalent
to").grid(column=1, row=2, sticky=E)
ttk.Label(mainframe,
text="meters").grid(column=3, row=2, sticky=W)

for child in mainframe.winfo_children():
child.grid_configure(padx=5, pady=5)

feet_entry.focus()
root.bind('<Return>', calculate)

root.mainloop()
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1610-GUI Programming

- File: Itse1359-1610.htm
- Published: 10/28/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1710-Change in Presentation Format

This module explains a change in presentation format.

Discussion

Previous modules in this collection have consisted of material written by Prof. Baldwin in combination with references to other freely available material on the web. Most of the **primary** modules were accompanied by a **review** module containing review questions and answers on the material written by Prof. Baldwin.

Beginning with the module titled [Itse1359-1810-Preface to Text Processing](#) each module will consist primarily of references to freely available material on the web along with review questions and answers pertaining to that material.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1710-Change in Presentation Format
- File: Itse1359-1710.htm
- Published: 11/10/14
- Revised: 01/22/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1720-Doctest

This module provides an introduction to the doctest module.

Table of contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [An interactive session](#)
 - [An automated interactive session](#)
 - [The test file named Py1359_1720_01](#)
 - [The batch the named Py1359_1720_01](#)
 - [The output](#)
 - [A more substantive example](#)
 - [The module named Py1359_1720_02](#)
 - [The test file named Py1359_1720_02](#)
 - [The batch file named Py1359_1720_02](#)
 - [The output](#)
 - [Embedding a test in a docstring](#)
 - [What is a docstring?](#)
 - [The module named Py1359_1720_03](#)
 - [The batch file named Py1359_1720_03](#)
 - [The output](#)
- [Run the program](#)
- [What's next?](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

In previous modules, you have learned the basics of Python programming, control flow, classes and objects, input/output, and GUI programming.

What you will learn

In this and the following module, you will learn how to use the Python **doctest** module to test your Python programs.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- [Figure 1](#). Simple interactive session.
- [Figure 2](#). Output produced by the test file named Py1359_1720_01.txt.
- [Figure 3](#). Output produced by the test file named Py1359_1720_02.txt.
- [Figure 4](#). Output produced by the test embedded in the docstring.

Listings

- [Listing 1](#). Contents of the test file named Py1359_1720_01.txt.
- [Listing 2](#). Contents of the file named Py1359_1720_01.bat.
- [Listing 3](#). Contents of the file named Py1359_1720_02.py.
- [Listing 4](#). Contents of the test file named Py1359_1720_02.txt.
- [Listing 5](#). Contents of the batch file named Py1359_1720_02.bat.
- [Listing 6](#). Contents of the file named Py1359_1720_03.py.
- [Listing 7](#). Contents of the batch file named Py1359_1720_03.bat.

General background information

According to [The Python Standard Library -- 26. Development Tools](#),

"The doctest and unittest modules contain frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced."

The material in this module is based heavily on [The Python Standard Library -- 26.2 doctest -- Test interactive Python examples](#). According to that document,

"The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown."

Stated differently, the **doctest** module can be used to automate the execution of code that you might otherwise execute in an interactive session. However, instead of having the computer present the result to you, you specify the required result and the computer tests the required result against the actual result and notifies you when the two fail to match.

There are at least **two different ways to use doctest** :

1. You can write the test code in a separate text file and use that file to perform the testing.
2. You can embed the test code in one or more **docstrings** in a module and cause the test code to be executed each time you execute the module as a script.

A simple example of each approach will be provided and explained in this module.

Discussion and sample code

An interactive session

[Figure 1](#) shows a simple interactive session run on the Python interactive command line interface.

Figure 1 . Simple interactive session.

```
>>> 2 + 3
5
>>>
```

If you have forgotten what the text in [Figure 1](#) means, see the earlier module titled *Itse1359-1010-Getting Started* .

An automated interactive session

This first example is provided solely to show how **doctest** works. It is not intended to be useful otherwise.

The test file named `Py1359_1720_01`

[Listing 1](#) shows the contents of a test file named `Py1359_1720_01.txt` .

Listing 1 . Contents of the test file named `Py1359_1720_01.txt`.

Listing 1 . Contents of the test file named Py1359_1720_01.txt.

```
>>> 2 + 3
6
```

Note that the contents of the test file mirror the input and the output of the interactive session shown in [Figure 1](#) except that the result of performing the computation was purposely specified incorrectly as 6 instead of 5. (When you create a **doctest** test file, you must specify the required output as shown in [Listing 1](#).)

Note that the **doctest** process is very picky regarding file names. Some file names that are valid for the operating system may not work when using **doctest** . However, if you restrict your file names to names that would be valid for variables, you should be okay.

[Listing 2](#) shows the contents of a Windows batch file named **Py1359_1720_01.bat** that I used to perform the test. Although not a requirement, the use of such a batch file makes it convenient to perform the test more than once with a minimal typing effort.

The batch the named Py1359_1720_01

Listing 2 . Contents of the file named Py1359_1720_01.bat.

```
echo off

rem set the path
path=%path%;"C:\Program Files (x86)\Python34"

rem perform the test
python -m doctest Py1359_1720_01.txt

pause
```

(Some of the text in [Listing 2](#) is peculiar to my machine. You can ignore the text shown in [Listing 2](#) down to the command that begins with the word **python** .)

The third command that begins with the word python in [Listing 2](#) executes the Python **doctest** module as a script and passes the name of the test file as a command-line argument to the module.

According to [Python v3.1.5 documentation -- Python Setup and Usage](#), "When called with -m module-name, the given module is located on the Python module path and executed as a script."

The syntax shown for the python command in [Listing 2](#) is a **command line shortcut** for calling the `testmod()` function in the `doctest` module. (*You will see more on this later.*)

The output

The execution of the python command in [Listing 2](#) produced the command line output shown in [Figure 2](#).

Figure 2 . Output produced by the test file named Py1359_1720_01.txt.

```
*****
File "Py1359_1720_01.txt", line 2, in Py1359_1720_01.txt
Failed example:
2 + 3
Expected:
6
Got:
5
*****
1 items had failures:
1 of 1 in Py1359_1720_01.txt
***Test Failed*** 1 failures.
```

[Figure 2](#) shows that the test failed because the result of the computation did not match the specified value of 6. In this case, the output from the computation was correct but the value specified by the test was incorrect.

When testing using the `doctest` module, you specify what the result must be for the test to pass. If the test fails to produce a matching result, the output will show that the test failed. For the case of [Listing 1](#) and [Listing 2](#), if the correct result had been specified, the test would simply have returned to the command prompt with no other output.

A verbose output could have been specified in [Listing 2](#) to cause the test to return positive results even if there were no failures. You will see how to do that in the next example.

A more substantive example

The module named `Py1359_1720_02`

This example will perform a test on a simple function named `sum` in a module named `Py1359_1720_02.py` as shown in [Listing 3](#).

Listing 3 . Contents of the file named `Py1359_1720_02.py`.

```
def sum(parA,parB):  
    return parA + parB
```

The test file named `Py1359_1720_02`

The test file for this example is shown in [Listing 4](#).

Listing 4 . Contents of the test file named `Py1359_1720_02.txt`.

```
>>> from Py1359_1720_02 import sum  
>>> sum(2,3)  
6
```

This file will actually cause two tests to be performed. The first test is to import the function named `sum` from the module named `Py1359_1720_02` . However, no visible output is expected from this test. Therefore, no output is specified in [Listing 4](#).

The second test is to call the function named `sum` (shown in [Listing 3](#)) passing 2 and 3 as parameters. The test file shows that an incorrect output value of 6 is expected from this test. *(As before, an incorrect value was specified to force the test to fail for illustration purposes only.)*

The batch file named `Py1359_1720_02`

The batch file used to execute the test is shown in [Listing 5](#).

Listing 5 . Contents of the batch file named Py1359_1720_02.bat.

```
echo off

rem set the path
path=%path%;"C:\Program Files (x86)\Python34"

rem perform the test
python -m doctest -v Py1359_1720_02.txt

pause
```

This batch file differs from the one shown in [Listing 2](#) in one respect. The **-v** switch following **doctest** in the command causes the output to be more verbose than the output from the previous example.

The output

The execution of the third (*python*) command in [Listing 5](#) produced the command line output shown in [Figure 3](#).

Figure 3 . Output produced by the test file named Py1359_1720_02.txt.

Figure 3 . Output produced by the test file named Py1359_1720_02.txt.

```
Trying:
from Py1359_1720_02 import sum
Expecting nothing
ok
Trying:
sum(2,3)
Expecting:
6
*****
File "Py1359_1720_02.txt", line 2, in Py1359_1720_02.txt
Failed example:
sum(2,3)
Expected:
6
Got:
5
*****
1 items had failures:
1 of 2 in Py1359_1720_02.txt
2 tests in 1 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

As mentioned earlier, this output is more verbose than the output shown in [Figure 2](#) . In this more verbose output, tests that pass are shown in addition to tests that fail. For example, the test of the **import** statement is shown as **ok** .

On the other hand, as expected, the call to the **sum** function is shown as **failed** . As before, both the location and the nature of the failure is explained.

Embedding a test in a docstring

As mentioned [earlier](#) , in addition to putting tests in separate text files, you can also embed tests in docstrings.

What is a docstring?

As shown in the earlier module titled *Itse1359-1270-Functions* , and described more fully in [What is a Docstring?](#) , a **docstring** is a string literal that occurs as the first statement in a module, function, class, or method definition. A docstring is surrounded by triple quotes.

The module named Py1359_1720_03

This example will perform a test on a simple function named **sum** in a module named **Py1359_1720_03.py** as shown in [Listing 6](#) .

Listing 6 . Contents of the file named Py1359_1720_03.py.

```
"""
>>> sum(2,3)
6
"""
def sum(parA,parB):
    return parA + parB

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

[Listing 6](#) is an update of the earlier module shown in [Listing 3](#). Test code was embedded in the docstring at the beginning of [Listing 6](#).

In addition, a special **if** statement was added at the end of [Listing 6](#) that calls the **testmod** function of the **doctest** module. (See the [earlier](#) comment regarding the *testmod* function and the command line shortcut.)

According to [The Python Standard Library -- 26.2.1. Simple Usage: Checking Examples in Docstrings](#), if **you include this code** at the end of a module, running the module as a script will cause tests that are embedded in docstrings to be executed.

Detailed information on the various options of the **testmod** function are provided at [The Python Standard Library -- 26.2.4. Basic API](#).

The batch file named Py1359_1720_03

The batch file used to execute the test is shown in [Listing 7](#).

Listing 7 . Contents of the batch file named Py1359_1720_03.bat.

Listing 7 . Contents of the batch file named Py1359_1720_03.bat.

```
echo off

rem set the path
path=%path%;"C:\Program Files (x86)\Python34"

rem perform the test
python Py1359_1720_03.py -v

pause
```

In this case there was no separate test file. Instead, the python command in [Listing 7](#) executed the module named **Py1359_1720_03.py** as a script triggering the test to be run as described [above](#).

The output

The **-v** switch in [Listing 7](#) causes the verbose version of the output to be produced as shown in [Figure 4](#). Once again, I set the test up in such a way as to cause it to fail for purposes of illustration.

Figure 4 . Output produced by the test embedded in the docstring.

Figure 4 . Output produced by the test embedded in the docstring.

```
Trying:
sum(2,3)
Expecting:
6
*****
File "Py1359_1720_03.py", line 4, in __main__
Failed example:
sum(2,3)
Expected:
6
Got:
5
1 items had no tests:
__main__.sum
*****
1 items had failures:
1 of 1 in __main__
1 tests in 2 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Run the program

I encourage you to copy the code provided in this module. Execute the code and confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

What's next?

Now that you know some of the mechanics of using **doctest** , you should study the next module in the collection titled *Itse1359-1720r-Review* . In addition, you should study some of the many articles and tutorials that are freely available on the web that provide insight into the use of **doctest** for unit testing. Links to a few of those documents are provided below. You can find many more with a simple web search.

- [26.2. doctest -- Test interactive Python examples](#)
- [Excerpt from book -- Testing in Python using doctest](#)
- [Excerpt from book -- Python: Unit Testing with Doctest](#)
- [Testing Your Code -- The Hitchhiker's Guide to Python](#)
- [Agile Testing -- Python unit testing part 1: the unittest module](#)
- [Agile Testing -- Python unit testing part 2: the doctest module](#)
- [Agile Testing -- Python unit testing part 3: the py.test tool and library](#)
- [Python3 Tutorial: Tests, DocTests, UnitTests -- Python Course](#)
- [An Extended Introduction to the nose Unit Testing Framework -- Running doctests in nose](#)
- [Doctests: run doctests with nose -- nose 1.3.4 documentation](#)
- [Unit Testing Python Using doctest - YouTube](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1720-Doctest Introduction
- File: Itse1359-1720.htm
- Published: 11/10/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1720r-Review

This module contains review questions and answers keyed to the module titled Itse1359-1720-Doctest as well as review questions and answers keyed to the document at <https://docs.python.org/3/library/doctest.html>.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1720-Doctest* . In addition, some of the questions and answers are keyed to the document titled [The Python Standard Library -- 26.2 doctest -- Test interactive Python examples](#) .

Once you study that material, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The test file shown in [Figure 1](#) , when executed using the command shown in [Figure 2](#) , produces the output shown in [Figure 3](#) .

Figure 1 . Question 1 test code named Py1359_1720r_01.txt.

Figure 1 . Question 1 test code named Py1359_1720r_01.txt.

```
Beginning of first test
>>> 2 + 3
5

End of first test
Beginning of second test
>>> 2 + 3
6

End of second test
```

Figure 2 . Question 1 execution command.

```
python -m doctest Py1359_1720r_01.txt
```

Figure 3 . Question 1 possible output.

Figure 3 . Question 1 possible output.

```
*****
File "Py1359_1720r_01.txt", line 2, in Py1359_1720r_01.txt
Failed example:
    2 + 3
Expected:
    6
Got:
    5
*****
File "Py1359_1720r_01.txt", line 7, in Py1359_1720r_01.txt
Failed example:
    2 + 3
Expected:
    6
Got:
    5
*****
1 items had failures:
  2 of   2 in Py1359_1720r_01.txt
***Test Failed*** 2 failures.
```

Go to [answer 1](#)

Question 2

True or False? The test file shown in [Figure 5](#), when executed using the command shown in [Figure 6](#), produces the output shown in [Figure 7](#).

Figure 5 . Question 2 test code named Py1359_1720r_02.txt.

```
Beginning of test
>>> 2 + 3
5
End of test
```

Figure 6 . Question 2 execution command.

```
python -m doctest Py1359_1720r_02.txt -v
```

Figure 7 . Question 2 possible output.

```
Trying:
    2 + 3
Expecting:
    5
ok
1 items passed all tests:
   1 tests in Py1359_1720r_02.txt
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
```

Go to [answer 2](#)

Question 3

True or False? Given the Python module shown in [Figure 9](#), when the test file shown in [Figure 10](#) is executed using the command shown in [Figure 11](#), it produces the output shown in [Figure 12](#).

Figure 9 . Question 3 Python module named Py1359_1720r_03.py.

```
def sum(parA, parB):
    return parA + parB

def product(parC, parD):
    return parC * parD
```

Figure 10 . Question 3 test code named Py1359_1720r_03.txt.

```
>>> from Py1359_1720r_03 import sum
>>> sum(2,3)
5
>>> from Py1359_1720r_03 import product
>>> product(2,3)
6
```

Figure 11 . Question 3 execution command.

```
python -m doctest -v Py1359_1720r_03.txt
```

Figure 12 . Question 3 possible output.

Figure 12 . Question 3 possible output.

```
Trying:
    from Py1359_1720r_03 import sum
Expecting nothing
ok
Trying:
    sum(2,3)
Expecting:
    5
ok
Trying:
    from Py1359_1720r_03 import product
Expecting nothing
ok
Trying:
    product(2,3)
Expecting:
    6
ok
1 items passed all tests:
  4 tests in Py1359_1720r_03.txt
4 tests in 1 items.
4 passed and 0 failed.
Test passed.
```

Go to [answer 3](#)

Question 4

True or False? Given the Python module shown in [Figure 13](#), execution of the module as a script using the command shown in [Figure 14](#) produces the output shown in [Figure 15](#).

Figure 13 . Question 4 Python module named Py1359_1720r_04.py.

Figure 13 . Question 4 Python module named Py1359_1720r_04.py.

```
"""
Begin first test
>>> sum(2,3)
5

Begin second test
>>> product(2,3)
6
"""
def sum(parA,parB):
    return parA + parB

def product(parC,parD):
    return parC * parD

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 14 . Question 4 execution command.

```
python Py1359_1720r_04.py -v
```

Figure 15 . Question 4 possible output.

Figure 15 . Question 4 possible output.

```
Trying:
    sum(2,3)
Expecting:
    5
ok
Trying:
    product(2,3)
Expecting:
    6
ok
2 items had no tests:
    __main__.product
    __main__.sum
1 items passed all tests:
    2 tests in __main__
2 tests in 3 items.
2 passed and 0 failed.
Test passed.
```

Go to [answer 4](#)

Question 5

True or False? Given the Python module shown in [Figure 16](#), execution of the module as a script using the command shown in [Figure 17](#) produces the output shown in [Figure 18](#).

Figure 16 . Question 5 Python module named Py1359_1720r_05.py.

Figure 16 . Question 5 Python module named Py1359_1720r_05.py.

```
"""
>>> divide(2,3)
0
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
"""
def divide(parA,parB):
    return parA//parB

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 17 . Question 5 execution command.

```
python Py1359_1720r_05.py -v
```

Figure 18 . Question 5 possible output.

Figure 18 . Question 5 possible output.

```
Trying:
    divide(2,3)
Expecting:
    0
ok
Trying:
    divide(2,0)
Expecting:
    Traceback (most recent call last):
      ...
    ZeroDivisionError: integer division or modulo by zero
ok
1 items had no tests:
    __main__.divide
1 items passed all tests:
    2 tests in __main__
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Go to [answer 5](#)

Question 6

True or False? Given the Python module shown in [Figure 19](#), execution of the module as a script using the command shown in [Figure 20](#) produces the output shown in [Figure 21](#).

Figure 19 . Question 6 Python module named Py1359_1720r_06.py.

Figure 19 . Question 6 Python module named Py1359_1720r_06.py.

```
"""
>>> function(11)
Traceback (most recent call last):
...
ValueError: par must be <= 10
"""
def function(par):
    if par > 10:
        raise ValueError("par must be <= 10")
    else:
        return par

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 20 . Question 6 execution command.

```
python Py1359_1720r_06.py -v
```

Figure 21 . Question 6 possible output.

Figure 21 . Question 6 possible output.

```
Trying:
    function(11)
Expecting:
    Traceback (most recent call last):
      ...
    ValueError: par must be <= 10
ok
1 items had no tests:
    __main__.function
1 items passed all tests:
    1 tests in __main__
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Go to [answer 6](#)

Question 7

True or False? Given the Python modules shown in [Figure 22](#) and [Figure 23](#) and the test file shown in [Figure 25](#), execution of the command shown in [Figure 24](#), produces the output shown in [Figure 26](#).

Figure 22 . Question 7 Python module named Py1359_1720r_07_runner.py.

```
import doctest
doctest.testfile(Py1359_1720r_07.txt)
```

Figure 23 . Question 7 Python module named Py1359_1720r_07.py.

Figure 23 . Question 7 Python module named Py1359_1720r_07.py.

```
def sum(parA,parB):  
    return parA + parB  
  
def product(parC,parD):  
    return parC * parD
```

Figure 24 . Question 7 execution command.

```
python Py1359_1720r_07_runner.py
```

Figure 25 . Question 7 test code named Py1359_1720r_07.txt.

```
>>> from Py1359_1720r_07 import sum  
>>> sum(2,3)  
5  
>>> from Py1359_1720r_07 import product  
>>> product(2,3)  
5
```

Figure 26 . Question 7 possible output.

Figure 26 . Question 7 possible output.

```
*****
File "Py1359_1720r_07.txt", line 5, in Py1359_1720r_07.txt
Failed example:
    product(2,3)
Expected:
    5
Got:
    6
*****
1 items had failures:
  1 of   4 in Py1359_1720r_07.txt
***Test Failed*** 1 failures.
```

Go to [answer 7](#)

Question 8

True or False? Given the Python module shown in [Figure 27](#) and the test file shown in [Figure 28](#), execution of the module as a script using the command shown in [Figure 29](#), produces the output shown in [Figure 30](#).

Figure 27 . Question 8 Python module named Py1359_1720r_08.py.

```
def sum(parA,parB):
    return parA + parB

def product(parC,parD):
    return parC * parD

if __name__ == "__main__":
    import doctest
    doctest.testfile("Py1359_1720r_08.txt")
```

Figure 28 . Question 8 test code named Py1359_1720r_08.txt.

```
>>> from Py1359_1720r_08 import sum
>>> sum(2,3)
5
>>> from Py1359_1720r_08 import product
>>> product(2,3)
6
```

Figure 29 . Question 8 execution command.

```
python Py1359_1720r_08.py -v
```

Figure 30 . Question 8 possible output.

Figure 30 . Question 8 possible output.

```
Trying:
    from Py1359_1720r_08 import sum
Expecting nothing
ok
Trying:
    sum(2,3)
Expecting:
    5
ok
Trying:
    from Py1359_1720r_08 import product
Expecting nothing
ok
Trying:
    product(2,3)
Expecting:
    6
ok
1 items passed all tests:
  4 tests in Py1359_1720r_08.txt
4 tests in 1 items.
4 passed and 0 failed.
Test passed.
```

Go to [answer 8](#)

Question 9

True or False? Given the Python version 3 module shown in [Figure 31](#), execution of the module as a script using the command shown in [Figure 32](#), produces the output shown in [Figure 33](#).

Figure 31 . Question 9 Python module named Py1359_1720r_09.py.

Figure 31 . Question 9 Python module named Py1359_1720r_09.py.

```
"""
>>> divide(1,3)
0.333333
"""
def divide(parA,parB):
    return parA/parB

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 32 . Question 9 execution command.

```
python Py1359_1720r_09.py -v
```

Figure 33 . Question 9 possible output.

```
Trying:
    divide(1,3)
Expecting:
    0.3333333333333333
ok
1 items had no tests:
    __main__.divide
1 items passed all tests:
   1 tests in __main__
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Figure index

- [Figure 1](#). Question 1 test code named Py1359_1720r_01.txt.
- [Figure 2](#). Question 1 execution command.
- [Figure 3](#). Question 1 possible output.
- [Figure 4](#). Question 1 actual output.
- [Figure 5](#). Question 2 test code named Py1359_1720r_02.txt.
- [Figure 6](#). Question 2 execution command.
- [Figure 7](#). Question 2 possible output.
- [Figure 8](#). Question 2 actual output.
- [Figure 9](#). Question 3 Python module named Py1359_1720r_03.py.
- [Figure 10](#). Question 3 test code named Py1359_1720r_03.txt.
- [Figure 11](#). Question 3 execution command.
- [Figure 12](#). Question 3 possible output.
- [Figure 13](#). Question 4 Python module named Py1359_1720r_04.py.
- [Figure 14](#). Question 4 execution command.
- [Figure 15](#). Question 4 possible output.
- [Figure 16](#). Question 5 Python module named Py1359_1720r_05.py.
- [Figure 17](#). Question 5 execution command.
- [Figure 18](#). Question 5 possible output.
- [Figure 19](#). Question 6 Python module named Py1359_1720r_06.py.
- [Figure 20](#). Question 6 execution command.
- [Figure 21](#). Question 6 possible output.
- [Figure 22](#). Question 7 Python module named Py1359_1720r_07_runner.py.
- [Figure 23](#). Question 7 Python module named Py1359_1720r_07.py.
- [Figure 24](#). Question 7 execution command.
- [Figure 25](#). Question 7 test code named Py1359_1720r_07.txt.
- [Figure 26](#). Question 7 possible output.
- [Figure 27](#). Question 8 Python module named Py1359_1720r_08.py.
- [Figure 28](#). Question 8 test code named Py1359_1720r_08.txt.
- [Figure 29](#). Question 8 execution command.
- [Figure 30](#). Question 8 possible output.
- [Figure 31](#). Question 9 Python module named Py1359_1720r_09.py.
- [Figure 32](#). Question 9 execution command.
- [Figure 33](#). Question 9 possible output.
- [Figure 34](#). Question 9 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 9

False. The actual output is shown in [Figure 34](#). This question was included to point out that testing floating point results can be problematic. Many floating point values are simply estimates. Although this example was purposely caused to fail, a given floating point computation may not produce exactly the same results on different platforms. Therefore a test that specifies a given floating point result may pass on one platform and fail on a different platform. [The Python Standard Library -- 26.2 doctest -- Test interactive Python examples](#) provides some suggestions for dealing with this issue.

Figure 34 . Question 9 actual output.

Figure 34 . Question 9 actual output.

```
Trying:
    divide(1,3)
Expecting:
    0.333333
*****
File "Py1359_1720r_09.py", line 4, in __main__
Failed example:
    divide(1,3)
Expected:
    0.333333
Got:
    0.3333333333333333
1 items had no tests:
    __main__.divide
*****
1 items had failures:
    1 of 1 in __main__
1 tests in 2 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Go back to [Question 9](#)

Answer 8

True. The call to **testfile** at the end of [Figure 27](#) illustrates another way to cause a test file to be executed.

Go back to [Question 8](#)

Answer 7

False. The output produced by the procedure described in this question includes the following error message:

NameError: name 'Py1359_1720r_07' is not defined

This indicates a syntax error in the specification of the name of the test file. While the concept is correct and it is possible to use Python code in a module similar to that shown in [Figure 22](#) to execute a test file such as that shown in [Figure 25](#), the name of the test file must be enclosed in quotation marks to be recognized as a test file. Enclosing the name of the test file in [Figure 22](#) in quotes would cause the procedure to produce the output shown in [Figure 26](#).

Go back to [Question 7](#)

Answer 6

True.

Go back to [Question 6](#)

Answer 5

True.

Go back to [Question 5](#)

Answer 4

True.

Go back to [Question 4](#)

Answer 3

True.

Go back to [Question 3](#)

Answer 2

False. The actual output is shown in [Figure 8](#). This is a formatting issue. If you include explanatory text in the text file, you must leave a blank line between the last line of specified output and the following line of explanatory text. Otherwise, the explanatory text will be included as part of required output as shown by the expected output value in [Figure 8](#).

Figure 8 . Question 2 actual output.

Figure 8 . Question 2 actual output.

```
Trying:
  2 + 3
Expecting:
  5
  End of test
*****
File "Py1359_1720r_02.txt", line 2, in Py1359_1720r_02.txt
Failed example:
  2 + 3
Expected:
  5
  End of test
Got:
  5
*****
1 items had failures:
  1 of   1 in Py1359_1720r_02.txt
1 tests in 1 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Go back to [Question 2](#)

Answer 1

False. The actual output is shown in [Figure 4](#). The test file contained two tests, one of which passed and the other of which failed. Verbose output was not specified. Therefore, the report didn't include information on the test that passed. The test file contained explanatory text, which is okay under certain formatting conditions, which I will get into in a subsequent question..

Figure 4 . Question 1 actual output.

Figure 4 . Question 1 actual output.

```
*****
File "Py1359_1720r_01.txt", line 7, in Py1359_1720r_01.txt
Failed example:
    2 + 3
Expected:
    6
Got:
    5
*****
1 items had failures:
  1 of  2 in Py1359_1720r_01.txt
***Test Failed*** 1 failures.
```

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1720r-Review
- File: Itse1359-1720r.htm
- Published: 11/10/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1810-Preface to Text Processing

This module is the preface for the Text Processing portion of the course.

Table of contents

- [Preface](#)
- [String operations](#)
- [Regular expressions](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

The learning resource for the **Text Processing** portion of the course consists of material from selected websites plus modules containing review questions and answers keyed to the material on those websites.

The **Text Processing** portion of the course consists of two main areas of study:

- String operations
- Regular expressions

A list of selected websites plus review questions keyed to those websites is provided for each area of study. The combined list of selected websites is:

- [The Python Standard Library: 6.1. string -- Common string operations](#)
- [The Python Standard Library: 6.2. re -- Regular expression operations](#)
- [The Python Standard Library: 2. Built-in Functions](#)
- [Python Library Reference -- 4.2.5 Match Objects](#)
- [Regular Expression HOWTO](#)
- [Charming Python: Text processing in Python](#)
- [tutorialspoint -- Python Strings](#)
- [tutorialspoint -- Python Regular Expressions](#)

- [Python Course -- Python 3 Tutorial -- Regular Expressions](#)
- [Python Course -- Python 3 Tutorial -- Advanced Regular Expressions](#)
- [Python Course -- Python 3 Tutorial -- Lambda, filter, reduce and map](#)
- [Python Notes \(0.14.0\) 9. The regular expression module](#)

Take-home programming assignments may be based on any material on these websites as well as material from any of the other modules in this collection of modules for teaching ITSE 1359. However, online tests for the **Text Processing** portion of the course will be restricted to topics addressed in the following two modules:

- Itse1359-1830-String Operations
- Itse1359-1850-Regular Expressions

String operations

The topics covered by the module titled *Itse1359-1830-String Operations* concentrates mainly on the various capabilities that Python offers for manipulating strings exclusive of the use of regular expressions. For example, this includes but is not limited to:

- typical indexing and slicing operations
- iterating on a string as a sequence using a **for** loop
- the **in** operator
- triple quoted strings
- raw strings
- strings in text files
- string methods

Regular expressions

According to [Wikipedia](#),

In theoretical computer science and formal language theory, a regular expression (abbreviated regex or regexp) and sometimes called a rational expression is a sequence of characters that forms

a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

The topic of regular expressions in Python is complex with two main threads:

1. How do you create a combination of characters that form a search pattern to accomplish a given objective?
2. **Given such a search pattern** , how to you use the Python programming language to carry out the search?

Creating the search pattern is pretty much the same no matter what programming language you are using. The character combinations used for creating the necessary search patterns are very similar from one programming language to the next. Since this is a course in Python programming and is not a course in regular expression search patterns, only the simple search patterns highlighted in the module titled *Itse1359-1850-Regular Expressions* will be included on tests. *(However there will be no such restriction on take-home assignments.)*

The main thrust of the ***Text Processing*** portion of this course insofar as regular expressions is concerned, will be on Item 2 [above](#) -- learning how to use the Python programming language to implement the effective use of regular expression search patterns.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1810-Preface to Text Processing
- File: Itse1359-1810.htm
- Published: 11/10/14

- Revised: 12/29/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1830-String Operations

This module contains review questions and answers keyed to instructional material on several websites.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to material on the following web sites:

- [The Python Standard Library: 6.1. string -- Common string operations](#)
- [Charming Python: Text processing in Python](#)
- [tutorialspoint -- Python Strings](#)

Once you study that material, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in [Figure 1](#) prints the single letter "r" (*without the quotation marks*).

Figure 1 . Question 1.

```
str = "This is a string."  
print(str[12])
```

Go to [answer 1](#)

Question 2

True or False? The code in the top panel of [Figure 2](#) produces the output shown in the bottom panel of [Figure 2](#).

Figure 2 . Question 2.

```
str = "This is a string."  
print(str)  
print(str[12])  
str[12] = "z"  
print(str)
```

```
=====  
This is a string.  
r  
This is a stzing.
```

Go to [answer 2](#)

Question 3

True or False? The code in the top panel of [Figure 4](#) produces the output shown in the bottom panel of [Figure 4](#).

Figure 4 . Question 3.

```
str = "This is a string."  
print(str)  
print(str[5:])
```

```
=====  
This is a string.  
is a string.
```

Go to [answer 3](#)

Question 4

True or False? The code in the top panel of [Figure 5](#) produces the output shown in the bottom panel of [Figure 5](#).

Figure 5 . Question 4.

```
str = "This is a string."  
print(str)  
print(str[5:-5])  
=====
```

This is a string.
a string.

Go to [answer 4](#)

Question 5

True or False? The code in the top panel of [Figure 7](#) produces the output shown in the bottom panel of [Figure 7](#).

Figure 7 . Question 5.

```
str = "This is a string."  
print(str)  
print(str[:4] + str[7:])  
=====
```

This is a string.
This a string.

Go to [answer 5](#)

Question 6

True or False? The code in the top panel of [Figure 8](#) produces the output shown in the bottom panel of [Figure 8](#).

Figure 8 . Question 6.

Figure 8 . Question 6.

```
str = "This is a string."  
for char in str:  
    print(char)  
=====
```

This is a string.

Go to [answer 6](#)

Question 7

True or False? The code in the top panel of [Figure 10](#) produces the output shown in the bottom panel of [Figure 10](#).

Figure 10 . Question 7.

```
str = "This is a string."  
chars = ["a", "s", "d", "f", "g", "h"]  
print(str)  
print(chars)  
for cnt in range(len(chars)):  
    if(chars[cnt]) not in str:  
        print(chars[cnt])  
=====
```

This is a string.
['a', 's', 'd', 'f', 'g', 'h']
a
s
g
h

Go to [answer 7](#)

Question 8

True or False? The code in the top panel of [Figure 12](#) produces the output shown in the bottom panel of [Figure 12](#).

Figure 12 . Question 8.

```
str = """Line 1
Line 2"""

print(str)
=====
Line 1
Line 2
```

Go to [answer 8](#)

Question 9

True or False? The code in the top panel of [Figure 13](#) produces the output shown in the bottom panel of [Figure 13](#).

Figure 13 . Question 9.

```
str = """Line 1\nLine 2"""

print(str)
=====
Line 1\nLine 2
```

Go to [answer 9](#)

Question 10

True or False? The code in the top panel of [Figure 15](#) produces the output shown in the bottom panel of [Figure 15](#).

Figure 15 . Question 10.

Figure 15 . Question 10.

```
str = r"""\nLine 1\nLine 2\n"""\nprint(str)\n=====\nLine 1\nLine 2
```

Go to [answer 10](#)

Question 11

True or False? Given a file named **1359-1830-11-input.txt** containing the text shown in the top panel of [Figure 17](#), the code in the middle panel of Figure 17 produces the output shown in the bottom panel of [Figure 17](#).

Figure 17 . Question 11.

```
Line 1\nLine 2\nLine 3\n=====\ninfile = open("1359-1830-11-input.txt")\nprint("Name of the file: ", infile.name)\n\nline = infile.readline()\nprint(line)\ninfile.close()\n=====\nName of the file: 1359-1830-11-input.txt\nLine 1
```

Go to [answer 11](#)

Question 12

True or False? Given a file named **1359-1830-12-input.txt** containing the text shown in the top panel of [Figure 18](#), the code in the middle panel of [Figure 18](#) produces the output shown in the bottom panel of [Figure 18](#).

Figure 18 . Question 12.

Line 1
Line 2
Line 3

```
=====
infile = open("1359-1830-12-input.txt")
print("Name of the file: ", infile.name)
for line in infile.readlines():
    print(line)
=====
```

```
Name of the file:  1359-1830-12-input.txt
Line 1

Line 2

Line 3
```

Go to [answer 12](#)

Question 13

True or False? Given a file named **1359-1830-13-input.txt** containing the text shown in the top panel of [Figure 19](#), the code in the middle panel of [Figure 19](#) produces the output shown in the bottom panel of [Figure 19](#).

Figure 19 . Question 13.

Line 1
Line 2
Line 3

```
=====
infile = open("1359-1830-13-input.txt")
print("Name of the file: ", infile.name)
for line in infile.readline():
    print(line)
=====
```

Line 1

Go to [answer 13](#)

Question 14

True or False? The code in the top panel of [Figure 21](#) produces the output shown in the bottom panel of [Figure 21](#).

Figure 21 . Question 14.

```
import string

print(string.digits)
print(string.hexdigits)
print(string.octdigits)
=====
01234567
0123456789
0123456789abcdefABCDEF
```

Go to [answer 14](#)

Question 15

True or False? The code in the top panel of [Figure 23](#) produces the output shown in the bottom panel of [Figure 23](#).

Figure 23 . Question 15.

```
str = "this is a string."

print(str.capitalize())
=====
THIS IS A STRING.
```

Go to [answer 15](#)

Figure index

- [Figure 1](#). Question 1.
- [Figure 2](#). Question 2.
- [Figure 3](#). Answer 2.
- [Figure 4](#). Question 3.
- [Figure 5](#). Question 4.
- [Figure 6](#). Answer 4.

- [Figure 7](#). Question 5.
- [Figure 8](#). Question 6.
- [Figure 9](#). Answer 6.
- [Figure 10](#). Question 7.
- [Figure 11](#). Answer 7.
- [Figure 12](#). Question 8.
- [Figure 13](#). Question 9.
- [Figure 14](#). Answer 9.
- [Figure 15](#). Question 10.
- [Figure 16](#). Answer 10.
- [Figure 17](#). Question 11.
- [Figure 18](#). Question 12.
- [Figure 19](#). Question 13.
- [Figure 20](#). Answer 13.
- [Figure 21](#). Question 14.
- [Figure 22](#). Answer 14.
- [Figure 23](#). Question 15.
- [Figure 24](#). Answer 15.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 15

False. The output is shown in [Figure 24](#).

Figure 24 . Answer 15.

```
This is a string.
```

Go back to [Question 15](#)

Answer 14

False. The answer is shown in [Figure 22](#).

Figure 22 . Answer 14.

```
0123456789
0123456789abcdefABCDEF
01234567
```

Go back to [Question 14](#)

Answer 13

False. The output is shown in [Figure 20](#). Students should analyze this code in order to explain the result.

Figure 20 . Answer 13.

Figure 20 . Answer 13.

```
Name of the file: 1359-1830-13-input.txt
L
i
n
e

1
```

Go back to [Question 13](#)

Answer 12

True.

Go back to [Question 12](#)

Answer 11

True.

Go back to [Question 11](#)

Answer 10

False. The output is shown in [Figure 16](#). Note the "r" to the left of the triple quote in [Figure 15](#). This causes the backslash in the string that might otherwise be interpreted as an escape character to be treated as a ordinary character.

Figure 16 . Answer 10.

```
Line 1\nLine 2
```

Go back to [Question 10](#)

Answer 9

False. The output is shown in [Figure 14](#). Note the newline escape characters in the string.

Figure 14 . Answer 9.

```
Line 1  
Line 2
```

Go back to [Question 9](#)

Answer 8

True.

Go back to [Question 8](#)

Answer 7

False. The output is shown in [Figure 11](#). Note the use of the **not** operator.

Figure 11 . Answer 7.

```
This is a string.  
['a', 's', 'd', 'f', 'g', 'h']  
d  
f
```

Go back to [Question 7](#)

Answer 6

False. The output is shown in [Figure 9](#).

Figure 9 . Answer 6.

Figure 9 . Answer 6.

```
T  
h  
i  
s  
  
i  
s  
  
a  
  
s  
t  
r  
i  
n  
g  
.
```

Go back to [Question 6](#)

Answer 5

True.

Go back to [Question 5](#)

Answer 4

False. The output is shown in [Figure 6](#). You can slice a string from the end using a negative index.

Figure 6 . Answer 4.

```
This is a string.  
is a st
```

Go back to [Question 4](#)

Answer 3

True.

Go back to [Question 3](#)

Answer 2

False. The actual output is shown in [Figure 3](#). A Python string is immutable. Therefore, you cannot modify a character within the string using simple indexing as shown in [Figure 2](#).

Figure 3 . Answer 2.

```
This is a string.  
r  
Traceback (most recent call last):  
  File "1359-1830-02.py", line 8, in <module>  
    str[12] = "z"  
TypeError: 'str' object does not support item assignment
```

Go back to [Question 2](#)

Answer 1

True. Individual characters in a string can be accessed using a zero-based index.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1830-String Operations
- File: Itse1359-1830.htm
- Published: 11/10/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1850-Regular Expressions

This module contains review questions and answers keyed to instructional material on several websites.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to material on the following web sites:

- [The Python Standard Library: 6.2. re -- Regular expression operations](#)
- [The Python Standard Library: 2. Built-in Functions](#)
- [Charming Python: Text processing in Python](#)
- [tutorialspoint -- Python Regular Expressions](#)
- [Regular Expression HOWTO](#)
- [Python Course -- Python 3 Tutorial -- Regular Expressions](#)
- [Python Course -- Python 3 Tutorial -- Advanced Regular Expressions](#)
- [Python Course -- Python 3 Tutorial -- Lambda, filter, reduce and map](#)
- [Python Notes \(0.14.0\) 9. The regular expression module](#)
- [Python Library Reference -- 4.2.5 Match Objects](#)

Once you study that material, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

Given: **RE** or **re** is an abbreviation for *Regular Expression* . Also, the primary module used for regular expressions in Python is named **re** . When using an RE pattern for matching characters, most letters and characters will simply match themselves. However, some characters are special *metacharacters* , and don't match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. (See [Regular Expression HOWTO](#).)

True or False? The following characters are included in the set of metacharacters:

[\$!

Go to [answer 1](#)

Question 2

True or False? The metacharacters [and] are used for specifying a character class, which is a set of characters that you wish to match.

Go to [answer 2](#)

Question 3

True or False? Within a character class, characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a three dashes as in [abc] or [a---c].

Go to [answer 3](#)

Question 4

True or False? The pattern [abc\$] will match any of the characters a, b, c, or \$.

Go to [answer 4](#)

Question 5

True or False? The pattern [^abc\$] will match any of the characters a, b, c, or \$.

Go to [answer 5](#)

Question 6

True or False? The code in the top panel of [Figure 2](#) produces the output shown in the bottom panel of [Figure 2](#).
(Hint: See http://www.python-course.eu/python3_lambda.php for lambda)

Figure 2 . Question 6.

Figure 2 . Question 6.

```
def prod(a,b):  
    return a*b  
  
def sum(c,d):  
    return c+d  
  
comp = lambda w,x,y,z : sum(w,x) * prod(y,z)  
print(comp(3,4,5,6))  
=====
```

210

Go to [answer 6](#)

Question 7

True or False? The code in the top panel of [Figure 3](#) produces the output shown in the bottom panel of [Figure 3](#).

Figure 3 . Question 7.

```
highNumbers = list(filter((lambda x: x > 5), range(10)))  
print(highNumbers)  
=====
```

[5, 6, 7, 8, 9]

Go to [answer 7](#)

Question 8

True or False? The code in the top panel of [Figure 5](#) produces the output shown in the bottom panel of [Figure 5](#).

Figure 5 . Question 8.

Figure 5 . Question 8.

```
import re
str = "abcd$"

if re.search(r'[ceg$]', str):
    print("Match")
else:
    print("No Match")

if re.search(r'^abcd$', str):
    print("Match")
else:
    print("No Match")
=====
Match
Match
```

Go to [answer 8](#)

Question 9

True or False? The code in the top panel of [Figure 7](#) produces the output shown in the bottom panel of [Figure 7](#).

Figure 7 . Question 9.

```
import re
s = "pqrabcdefg-abcd-abc$"

matches = re.findall(r'ab', s)
for substr in matches:
    print(substr)

searchObj = re.search(r'ab', s)
if searchObj:
    print(str(searchObj.start()) + "-" + str(searchObj.end()))
=====
ab
ab
ab
3-5
```

Go to [answer 9](#)

Question 10

True or False? The code in the top panel of [Figure 8](#) produces the output shown in the bottom panel of [Figure 8](#).

Figure 8 . Question 10.

```
myList = [0,1,2,3,4,5,6]
print(myList)
print("ok")
print(*myList)
```

```
=====
[0, 1, 2, 3, 4, 5, 6]
ok
[0, 1, 2, 3, 4, 5, 6]
```

Go to [answer 10](#)

Question 11

Note: You will need to understand **lambda** and **filter** to understand the code in most of the remaining questions.

True or False? The code in the top panel of [Figure 10](#) produces the output shown in the bottom panel of [Figure 10](#).

Figure 10 . Question 11.

Figure 10 . Question 11.

```
import re
myList = ('Dick', 'TomDickHarry', 'Dick is my name')

for testString in myList:
    print("Test string:",testString)
    lambdaFunction = lambda arg: re.search(r"Dick", arg)
    matchObject = lambdaFunction(testString)
    if(matchObject!=None):
        print(" Matching subString:",matchObject.group())
=====
Test string: Dick
    Matching subString: Dick
Test string: TomDickHarry
    Matching subString: Dick
Test string: Dick is my name
    Matching subString: Dick
```

Go to [answer 11](#)

Question 12

True or False? The code in the top panel of [Figure 11](#) produces the output shown in the bottom panel of [Figure 11](#).

Figure 11 . Question 12.

```
import re
myList = ('Dick', 'TomDickHarry', 'Dick is my name')

for testString in myList:
    print("Test string:",testString)
    lambdaFunction = lambda arg: re.match(r"Dick", arg)
    matchObject = lambdaFunction(testString)
    if(matchObject!=None):
        print(" Matching subString:",matchObject.group())
=====
Test string: Dick
    Matching subString: Dick
Test string: TomDickHarry
    Matching subString: Dick
Test string: Dick is my name
    Matching subString: Dick
```


Go to [answer 12](#)

Question 13

True or False? The code in the top panel of [Figure 13](#) produces the output shown in the bottom panel of [Figure 13](#).

Figure 13 . Question 13.

```
import re
myList = ('Dick', 'TomDickHarry', 'Dick is my name')

for testString in myList:
    print("Test string:",testString)
    matchObject = re.match(r"Dick", testString)
    if(matchObject!=None):
        print("    Matching subString:",matchObject.group())
=====
Test string: Dick
    Matching subString: Dick
Test string: TomDickHarry
Test string: Dick is my name
    Matching subString: Dick
```

Go to [answer 13](#)

Question 14

True or False? The code in the top panel of [Figure 14](#) produces the output shown in the bottom panel of [Figure 14](#).

Figure 14 . Question 14.

```
import re
myList = ('Joe is your name ', 'Dick ', 'TomDickHarry ', 'Dick is my name ')
filterObj = filter((lambda arg: re.search(r"Dick", arg)), myList)
print(*filterObj)
=====
Joe is your name   Dick   TomDickHarry   Dick is my name
```

Go to [answer 14](#)

Question 15

True or False? The code in the top panel of [Figure 16](#) produces the output shown in the bottom panel of [Figure 16](#).

Figure 16 . Question 15.

```
import re
myList = ('Joe is your name ', 'Dick ', 'TomDickHarry ', 'Dick is my name ')
filterObj = filter((lambda arg: re.match(r"Dick", arg)), myList)
print(*filterObj)
=====
Dick    Dick is my name
```

Go to [answer 15](#)

Figure index

- [Figure 1](#). Answer 1.
- [Figure 2](#). Question 6.
- [Figure 3](#). Question 7.
- [Figure 4](#). Answer 7.
- [Figure 5](#). Question 8.
- [Figure 6](#). Answer 8.
- [Figure 7](#). Question 9.
- [Figure 8](#). Question 10.
- [Figure 9](#). Answer 10.
- [Figure 10](#). Question 11.
- [Figure 11](#). Question 12.
- [Figure 12](#). Answer 12.
- [Figure 13](#). Question 13.
- [Figure 14](#). Question 14.
- [Figure 15](#). Answer 14.
- [Figure 16](#). Question 15.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 15

True.

Go back to [Question 15](#)

Answer 14

False. The output is shown in [Figure 15](#). The first string in **myList** in [Figure 14](#) did not match the **re** pattern. Therefore, it did not appear in the output.

Figure 15 . Answer 14.

```
Dick    TomDickHarry    Dick is my name
```

Go back to [Question 14](#)

Answer 13

True. Note the similarity between the code in this question and the code in Question 11 and Question 12. The **lambda** function was used in those earlier questions simply to show how to use regular expressions with a **lambda** function. This will be important for use with the **filter** function later.

Go back to [Question 13](#)

Answer 12

False. The output is shown in [Figure 12](#). Note the difference in the behavior of the **search** and **match** functions of the **re** module.

Figure 12 . Answer 12.

```
Test string: Dick
Matching subString: Dick
Test string: TomDickHarry
Test string: Dick is my name
Matching subString: Dick
```

Go back to [Question 12](#)

Answer 11

True.

Go back to [Question 11](#)

Answer 10

False. The output is shown in [Figure 9](#). Note the effect of the asterisk ***** in the argument list of the call to the **print** function in [Figure 8](#). This will be important in another question later in this module.

Figure 9 . Answer 10.

Figure 9 . Answer 10.

```
[0, 1, 2, 3, 4, 5, 6]
ok
0 1 2 3 4 5 6
```

Go back to [Question 10](#)

Answer 9

True.

Go back to [Question 9](#)

Answer 8

False. The output is shown in [Figure 6](#). Note the complement operator ^ at the left end of the class in the second **if** statement in [Figure 5](#).

Figure 6 . Answer 8.

```
Match
No Match
```

Go back to [Question 8](#)

Answer 7

False. The output is shown in [Figure 4](#). See an explanation of **lambda** and **filter** at [Python Course -- Python 3 Tutorial -- Lambda, filter, reduce and map](#). See **list** at The [Python Standard Library: 2. Built-in Functions -- class list](#)

Figure 4 . Answer 7.

Figure 4 . Answer 7.

[6, 7, 8, 9]

Go back to [Question 7](#)

Answer 6

True. See an explanation of **lambda** at [Python Course -- Python 3 Tutorial -- Lambda, filter, reduce and map](#).

Go back to [Question 6](#)

Answer 5

False. The pattern **[^abc\$]** will match any character *other than* a, b, c, or \$. Note the ^ character at the beginning and see [Regular Expression HOWTO](#).

Go back to [Question 5](#)

Answer 4

True. Although \$ is usually a metacharacter, metacharacters are not active inside classes. Inside a class, \$ is treated as an ordinary character.

Go back to [Question 4](#)

Answer 3

False. Within a character class, characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a **single** dash as in [abc] or [a-c]. See [Regular Expression HOWTO](#).

Go back to [Question 3](#)

Answer 2

True.

Go back to [Question 2](#)

Answer 1

False. The metacharacters are shown in [Figure 1](#) and do not include the exclamation character '!'. See [Regular Expression HOWTO](#).

Figure 1 . Answer 1.

. ^ \$ * + ? { } [] \ | ()

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1850-Regular Expressions
- File: Itse1359-1850.htm
- Published: 11/10/14
- Revised: 03/24/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1900-Preface to Networking and Databases

This module is the preface for the Networking and Databases portion of the course.

Table of contents

- [Preface](#)
- [Networking with HTTP](#)
- [Dbm and shelve databases](#)
- [SQLite database](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

The learning resource for the **Networking and Databases** portion of the course consists of material from selected websites plus modules containing review questions and answers keyed to the material on those websites.

The **Networking and Databases** portion of the course consists of three main areas of study:

- Networking with HTTP
- Dbm and shelve databases
- SQLite database

A list of selected websites plus review questions keyed to those websites is provided for each area of study. The combined list of selected websites is:

- [tutorialspoint - HTTP - Quick Guide](#)
- [w3.org - 3 Protocol Parameters](#)
- [The Python Standard Library - 21.12. http.client - HTTP protocol client](#)

- [The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"](#)
- [The Python Standard Library - 12.3. shelve - Python object persistence](#)
- [The Python Standard Library - 12.1. pickle - Python object serialization](#)
- [The Python Standard Library - 4.10. Mapping Types - dict](#)
- [W3Schools SQL Tutorial](#)
- [The Python Standard Library - 12.6. SQLite3 - DB-API 2.0 interface for SQLite databases](#)
- [Python Central - Introduction to SQLite in Python](#)
- [udemy - SQLite for Beginners](#)
- [tutorialspoint - SQLite Tutorial](#)
- [ZetCode - SQLite tutorial](#)

Take-home programming assignments may be based on any material on these websites as well as material from any of the other modules in this collection of modules for teaching ITSE 1359. However, online tests for the **Networking and Databases** portion of the course will be restricted to topics addressed in the following three modules:

- Itse1359-1910-Networking with HTTP
- Itse1359-1930-Dbm and Shelve Databases
- Itse1359-1950-SQLite Database

Networking with HTTP

Networking is a very broad topic. There are many textbooks and websites that discuss Python programs used for a variety of networking purposes, such as online chat, shared whiteboards, FTP, sockets, server sockets, etc. However, in order to narrow the topic somewhat, this course and the topics covered by the module titled *Networking with HTTP* concentrate mainly on what is probably the most common form of networking: HTTP.

According to [tutorialspoint - HTTP - Quick Guide](#),

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web (i.e.. internet) since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extension of its request methods, error codes and headers.

In other words, if you are reading this document online, you are probably accessing and reading it from a website using HTTP.

Dbm and shelve databases

In layman's terms, a database is a structure that can maintain data on a persistent basis. In other words, the data does not go away when the program that is accessing that data terminates. The same or other programs can be started up later to access and manipulate the data in the database.

Python supports several different types of databases, one of which is [DBM](#). However, configuring a system to support a particular type of database can be very tedious. For compatibility with different student's computing systems, this course will concentrate on a standard [Portable DBM implementation](#) known as [dbm.dumb](#).

According to [Portable DBM implementation](#), the **dbm.dumb** module is intended as a last resort fallback for the dbm module when a more robust module is not available. The **dbm.dumb** module is not written for speed and is not nearly as heavily used as the other database modules.

The **dbm.dumb** module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as dbm.gnu no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

A [shelf](#) is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the [pickle](#) module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

Aside from this brief description, this course will not address [shelve](#).

SQLite database

SQL stands for **Structured Query Language**. SQL is used to communicate with a database. According to ANSI (*American National Standards Institute*), it is the standard language for relational database management systems.

According to [The Python Standard Library - 12.6. sqlite3 - DB-API 2.0 interface for SQLite databases](#),

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language.

SQL is also a very broad topic. Considerable time, effort, and experience is required to master SQL. As evidence of this fact, see the somewhat voluminous [W3Schools SQL Tutorial](#).

Knowledge of SQL is a prerequisite for understanding SQLite. It is unreasonable to expect a student to learn SQL during a small portion of a one-semester course in Python. Therefore, the module titled *Itse1359-1950-SQLite Database* will barely scratch the surface and will provide only a cursory introduction to the use of SQLite.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1900-Preface to Networking and Databases
- File: Itse1359-1900.htm
- Published: 11/11/14
- Revised: 06/23/17

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1910-Networking with HTTP

This module contains review questions and answers keyed to instructional material on several websites.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to material on the following web sites:

- [tutorialspoint - HTTP - Quick Guide](#)
- [w3.org - 3 Protocol Parameters](#)
- [The Python Standard Library - 21.12. http.client - HTTP protocol client](#)

Once you study that material, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? HTTP is a TCP/IP based communication protocol, which is used to deliver data (HTML files, image files, query results etc.) on the World Wide Web. The default port is TCP 80, but other ports can be used. It provides a standardized way for computers to communicate with each other.

Go to [answer 1](#)

Question 2

True or False? With HTTP, a connection between the client and the server is maintained throughout the communication session covering multiple web pages.

Go to [answer 2](#)

Question 3

True or False? HTTP is media independent.

Go to [answer 3](#)

Question 4

True or False? HTTP is a stateless protocol.

Go to [answer 4](#)

Question 5

True or False? The HTTP protocol is a request/response protocol based on client/server based architecture where web browsers, robots, and search engines, etc. act like HTTP clients and a Web server acts as server.

Go to [answer 5](#)

Question 6

True or False? An http URL has the general form given below where the items in the square brackets are optional. If the port is empty or not given, port 8080 is assumed.

`http_URL = "http:" "/" host [":" port] [abs_path ["?" query]]`

Go to [answer 6](#)

Question 7

True or False? Assuming that the connection to the file named **index.html** at "www.dickbaldwin.com" is successful, the code in the top panel of [Figure 1](#) produces the output shown in the bottom panel of [Figure 1](#).

Figure 1 . Question 7.

```
import http.client

connection = http.client.HTTPConnection("www.dickbaldwin.com")
connection.request("GET", "/index.html")
response = connection.getresponse()
print("Status:", response.status, "Reason:", response.reason)

connection.close()
=====
Status: 200 Reason: OK
```

Go to [answer 7](#)

Question 8

True or False? Assuming that the connection to the file named **index.html** at "www.dickbaldwin.com" is successful, the code in the top panel of [Figure 2](#) produces the output shown in the bottom panel of [Figure 2](#) with minor variations that change from one run to the next.

Figure 2 . Question 8.

```
import http.client

connection = http.client.HTTPConnection("www.dickbaldwin.com")
connection.request("GET", "/index.html")
response = connection.getresponse()
print("Status:", response.status, "Reason:", response.reason)
print()
print(response.getheaders())

connection.close()
=====
Status: 200 Reason: OK

Server: Apache/2.2
Content-Type: text/html; charset=iso-8859-1
Date: Sat, 08 Nov 2014 22:08:37 GMT
Content-Language: en-US
Accept-Ranges: bytes
Connection: Keep-Alive
Set-Cookie: X-Mapping-bffmijpk=2C686CCC2A6865C34A535895FE033586; path=/
Content-Length: 3285
```

Go to [answer 8](#)

Question 9

True or False? Assuming that the connection to the file named **index.html** at "www.dickbaldwin.com" is successful, the code in the top panel of [Figure 4](#) produces the output shown in the bottom panel of [Figure 4](#) with minor variations that change from one run to the next.

Figure 4 . Question 9.

```
import http.client

connection = http.client.HTTPConnection("www.dickbaldwin.com")
connection.request("GET", "/index.html")
response = connection.getresponse()
print("Status:", response.status, "Reason:", response.reason)
print()
print(response.msg)

connection.close()
=====
Status: 200 Reason: OK

Server: Apache/2.2
Content-Type: text/html; charset=iso-8859-1
Date: Sat, 08 Nov 2014 22:15:55 GMT
Content-Language: en-US
Accept-Ranges: bytes
Connection: Keep-Alive
Set-Cookie: X-Mapping-bffmijpk=A6FAC0A1A16DBD577D9763B0D6D7684E; path=/
Content-Length: 3285
```

Go to [answer 9](#)

Question 10

True or False? Assuming that the connection to the file named **index.html** at "www.dickbaldwin.com" is successful, the code in the top panel of [Figure 5](#) produces the output shown in the bottom panel of [Figure 5](#) except that the actual date and time will change from one run to the next.

Figure 5 . Question 10.

Figure 5 . Question 10.

```
import http.client

connection = http.client.HTTPConnection("www.dickbaldwin.com")
connection.request("GET", "/index.html")
response = connection.getresponse()
print("Status:", response.status, "Reason:", response.reason)
print()
print(response.getheader('Date'))

connection.close()
=====
Status: 200 Reason: OK

Sat, 08 Nov 2014 22:25:58 GMT
```

Go to [answer 10](#)

Question 11

True or False? Assuming that the connection to the file named **index.html** at "www.dickbaldwin.com" is successful, the code in the top panel of [Figure 6](#) produces the output shown in the bottom panel of [Figure 6](#) except that the contents of the file named **index.html** may change over time.

Figure 6 . Question 11.

Figure 6 . Question 11.

```
import http.client

connection = http.client.HTTPConnection("www.dickbaldwin.com")
connection.request("GET", "/index.html")
response = connection.getresponse()
print("Status:", response.status, "Reason:", response.reason)
print()
print(response.read(410))

connection.close()
=====
Status: 200 Reason: OK

b'<?xml version="1.0" encoding="iso-8859-1"?>\r\n<!DOCTYPE html PUBLIC
"-//W3C//
DTD XHTML 1.0 Transitional//EN"\r\n "http://www.w3.org/TR/xhtml1/DTD/xhtml1
-transitional.dtd">\r\n<html
xmlns="http://www.w3.org/1999/xhtml">\r\n<head>\r\n
  <meta http-equiv="content-type" content="text/html; charset=iso-8859-1"
/>\r\n
  <title>Flex/ActionScript/C++/Scratch/Alice/C#/Java/JavaScript/XML\r\n Prog
ming, by Richard G Baldwin'
```

Go to [answer 11](#)

Figure index

- [Figure 1](#). Question 7.
- [Figure 2](#). Question 8.
- [Figure 3](#). Answer 8.
- [Figure 4](#). Question 9.
- [Figure 5](#). Question 10.
- [Figure 6](#). Question 11.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 11

True.

Go back to [Question 11](#)

Answer 10

True.

Go back to [Question 10](#)

Answer 9

True. This is also the code that produced the output with the format shown in the bottom panel of [Figure 2](#).

Go back to [Question 9](#)

Answer 8

False. Assuming that the connection to the file named **index.html** at "www.dickbaldwin.com" is successful, the code in the top panel of [Figure 2](#) produces the output shown in the bottom panel of [Figure 3](#) with minor variations that change from one run to the next. Note that the data values are the same as in the bottom panel of [Figure 2](#) but the format is different.

Figure 3 . Answer 8.

Status: 200 Reason: OK

```
[('Server', 'Apache/2.2'), ('Content-Type', 'text/html; charset=iso-8859-1'), ('Date', 'Sat, 08 Nov 2014 22:08:37 GMT'), ('Content-Language', 'en-US'), ('Accept-Ranges', 'bytes'), ('Connection', 'Keep-Alive'), ('Set-Cookie', 'X-Mapping-bffmijpk=2C686CCC2A6865C34A535895FE033586; path=/'), ('Content-Length', '3285')]
```

Go back to [Question 8](#)

Answer 7

True. See [The Python Standard Library - 21.12. http.client - HTTP protocol client](#).

Go back to [Question 7](#)

Answer 6

False. If the port is empty or not given, port 80 (not port 8080) is assumed. See section 3.2.2 of [w3.org - 3 Protocol Parameters](#).

Go back to [Question 6](#)

Answer 5

True.

Go back to [Question 5](#)

Answer 4

True. HTTP is a connectionless and as a result, HTTP is a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each other insofar as the protocol is concerned. Note however that browsers use various methods, such as cookies, to store information about previous connections but that capability is not built into the HTTP protocol.

Go back to [Question 4](#)

Answer 3

True. Any type of data can be sent by HTTP as long as both the client and server know how to handle the data content.

Go back to [Question 3](#)

Answer 2

False. The HTTP client (typically a browser) initiates an HTTP request. After a request is made, the client disconnects from the server and waits for a response. The server processes the request and re-establish the connection with the client to send response back.

Go back to [Question 2](#)

Answer 1

True. See [tutorialspoint -- HTTP -- Quick Guide](#).

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1910-Networking with HTTP
- File: Itse1359-1910.htm
- Published: 11/10/14
- Revised: 03/04/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor

do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1930-Dbm and Shelve Databases

This module contains review questions and answers keyed to instructional material on several websites.

Table of contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#)
- [Figure index](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to material on the following web sites:

- [The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"](#)
- [The Python Standard Library - 12.3. shelve - Python object persistence](#)
- [The Python Standard Library - 12.1. pickle - Python object serialization](#)
- [The Python Standard Library - 4.10. Mapping Types - dict](#)

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The webpage at [The Python Standard Library - 12.1. pickle - Python object serialization](#) contains the following warning:

Warning: The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Go to [answer 1](#)

Question 2

True or False? The webpage at [The Python Standard Library - 12.3. shelve - Python object persistence](#) contains the following warning:

Warning: Because the shelve module is backed by pickle, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Go to [answer 2](#)

Question 3

True or False? The webpage at [The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"](#) contains the following warning:

Warning: The dbm module is not intended to be secure against erroneous or maliciously constructed data. Never use dbm data received from an untrusted or unauthenticated source.

Go to [answer 3](#)

Question 4

True or False? **dbm** is a generic interface to variants of the DBM databases - **dbm.gnu** or **dbm.ndbm** . If neither of these modules is installed, the program will raise a "Module not found" error.

Go to [answer 4](#)

Question 5

True or False? The function named **dbm.whichdb(filename)** can be called to determine which database module should be used to open a given database file.

Go to [answer 5](#)

Question 6

True or False? The function named **dbm.getDatabase** can be called to open an existing **dbm** database file or to create a new **dbm** database file.

Go to [answer 6](#)

Question 7

True or False? The object returned by **dbm.open** supports the same basic functionality as tuples. Keys and their corresponding values can be stored, retrieved, and deleted. The **in** operator and the **keys** method are available, as well as **get** and **setdefault** .

Go to [answer 7](#)

Question 8

True or False? When using the **dbm.dumb** module, attempting to store a key or value that is not a string will raise an exception.

Go to [answer 8](#)

Question 9

True or False? The following statement appears on the webpage at [The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"](#).

Note: The dbm.dumb module is intended as a last resort fallback for the dbm module when a more robust module is not available. The dbm.dumb module is not written for speed and is not nearly as heavily used as the other database modules.

Go to [answer 9](#)

Question 10

True or False? The function named **dbm.dumb.open** can be called to create or open a **dumbdbm** database and return a **dumbdbm** object.

Go to [answer 10](#)

Question 11

True or False? The code in the top panel of [Figure 1](#) produces the output shown in the bottom panel of [Figure 1](#).

Figure 1 . Question 11.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

dumbDb[b'aNumber'] = 1234
print(dumbDb.get('aNumber'))

dumbDb.close()
=====
1234
```

Go to [answer 11](#)

Question 12

True or False? The code in the top panel of [Figure 3](#) produces the output shown in the bottom panel of [Figure 3](#).

Figure 3 . Question 12.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

dumbDb[b'aNumber'] = b'1234'
print(dumbDb.get('aNumber'))

dumbDb.close()
=====
b'1234'
```

Go to [answer 12](#)

Question 13

True or False? The code in the top panel of [Figure 4](#) produces the output shown in the bottom panel of [Figure 4](#).

Figure 4 . Question 13.

Figure 4 . Question 13.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

empName = b'Joe'

dumbDb[b'name'] = empName
dumbDb[b"cubicle"] = b"639"
dumbDb[b"phone"] = b"617.665.2154"

if(empName in dumbDb):
    print('name: ', dumbDb.get(b'name'))
    print('cubicle: ', dumbDb.get(b'cubicle'))
    print('phone: ', dumbDb.get(b'phone'))
else:
    print("Not in dumbDb")

dumbDb.close()
=====
name: b'Joe'
cubicle: b'639'
phone: b'617.665.2154'
```

Go to [answer 13](#)

Question 14

True or False? The code in the top panel of [Figure 6](#) produces the output shown in the bottom panel of [Figure 6](#).

Figure 6 . Question 14.

Figure 6 . Question 14.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

empName = b'Joe'

dumbDb[b'name'] = empName
dumbDb[b"cubicle"] = b"639"
dumbDb[b"phone"] = b"617.665.2154"

if(b'name' in dumbDb):
    print('name: ', dumbDb.get(b'name'))
    print('cubicle: ', dumbDb.get(b'cubicle'))
    print('phone: ', dumbDb.get(b'phone'))
else:
    print("Not in dumbDb")

dumbDb.close()
=====
name:  b'Joe'
cubicle:  b'639'
phone:  b'617.665.2154'
```

Go to [answer 14](#)

Question 15

True or False? The code in the top panel of [Figure 7](#) produces the output shown in the bottom panel of [Figure 7](#).

Figure 7 . Question 15.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

#Create keys and values
myKeys = b'name',b'cubicle',b'phone',b'age',b'gender',b'marital'
myValues = b'Joe',b'639',b'333.444.5555',b'39',b'male',b'married'

#Populate the database
for cnt in range(len(myKeys)):
    dumbDb[myKeys[cnt]] = myValues[cnt]

print('Get and display values')
values = dumbDb.values()
for value in values:
    print(value)
```

```

print()#blank line
print('Get and display keys')
keys = dumbDb.keys()
for key in keys:
    print(key)

print()
print('Attempt to pop and display two values')
print(dumbDb.pop(b'phone', 'no phone'))
print(dumbDb.pop(b'weight', 'no weight'))

print()
print('Get and display keys and values')
print('Database length', len(dumbDb))
items = dumbDb.items()
for item in items:
    print(item)

print()
print('Delete a key/value pair and display again')
del dumbDb[b'cubicle']
print('Database length', len(dumbDb))
items = dumbDb.items()
for item in items:
    print(item)

dumbDb.close()
=====
===
Get and display values
b'name'
b'age'
b'phone'
b'marital'
b'cubicle'
b'gender'

Get and display keys
b'Joe'
b'39'
b'333.444.5555'
b'married'
b'639'
b'male'

Attempt to pop and display two values
b'333.444.5555'
no weight

Get and display keys and values
Database length 5
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'cubicle', b'639')
(b'gender', b'male')

```

```
Delete a key/value pair and display again
Database length 4
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'gender', b'male')
```

Go to [answer 15](#)

Figure index

- [Figure 1](#). Question 11.
- [Figure 2](#). Answer 11.
- [Figure 3](#). Question 12.
- [Figure 4](#). Question 13.
- [Figure 5](#). Answer 13.
- [Figure 6](#). Question 14.
- [Figure 7](#). Question 15.
- [Figure 8](#). Answer 15.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 15

False. The output is shown in [Figure 8](#). The **keys** and the **values** were switched in the bottom panel of [Figure 7](#).

Figure 8 . Answer 15.

```
Get and display values
b'Joe'
b'39'
b'333.444.5555'
b'married'
b'639'
b'male'

Get and display keys
b'name'
b'age'
b'phone'
b'marital'
b'cubicle'
b'gender'

Attempt to pop and display two values
b'333.444.5555'
no weight

Get and display keys and values
Database length 5
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'cubicle', b'639')
(b'gender', b'male')

Delete a key/value pair and display again
Database length 4
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'gender', b'male')
```

Go back to [Question 15](#)

Answer 14

True.

Go back to [Question 14](#)

Answer 13

False. The output is shown in [Figure 5](#). The **in** operator searches for a match in the **keys** and does not search for a match in the **values**.

Figure 5 . Answer 13.

```
Not in dumbDb
```

Go back to [Question 13](#)

Answer 12

True.

Go back to [Question 12](#)

Answer 11

False. The code in [Figure 1](#) produces an error by attempting to store a number as a value. The output (*with some text deleted for brevity*) is shown in [Figure 2](#).

Figure 2 . Answer 11.

```
Traceback (most recent call last):
  File "1359-1930-11.py", line 11, in <module>
    dumbDb[b'aNumber'] = 1234
...
    raise TypeError("values must be bytes or strings")
TypeError: values must be bytes or strings
```

Go back to [Question 11](#)

Answer 10

True. See [The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"](#) .

Go back to [Question 10](#)

Answer 9

True as of 11/09/2014.

Go back to [Question 9](#)

Answer 8

True. See comments in sample code at [The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"](#) .

Go back to [Question 8](#)

Answer 7

False. The object returned by **dbm.open** supports the same basic functionality as *dictionaries* . Keys and their corresponding values can be stored, retrieved, and deleted. The **in** operator and the **keys** method are available, as well as **get** and **setdefault** .

Go back to [Question 7](#)

Answer 6

False. The function named **dbm.open** can be called to open an existing **dbm** database file or to create a new **dbm** database file.

Go back to [Question 6](#)

Answer 5

True.

Go back to [Question 5](#)

Answer 4

False. **dbm** is a generic interface to variants of the DBM databases - **dbm.gnu** or **dbm.ndbm** . If neither of these modules is installed, the slow-but-simple implementation in module **dbm.dumb** will be used.

Go back to [Question 4](#)

Answer 3

False. As of 11/09/2014, no such warning appears on the referenced webpage. Therefore, it might be concluded that even though **shelve** has features that are not available in **dbm**, **dbm** is the more secure of the two. As a result, this course will concentrate on **dbm** and ignore **shelve**.

Go back to [Question 3](#)

Answer 2

True as of 11/09/2014.

Go back to [Question 2](#)

Answer 1

True as of 11/09/2014.

Go back to [Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1930-Dbm and Shelve Databases
- File: Itse1359-1930.htm
- Published: 11/10/14
- Revised: 07/28/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1950-SQLite Database

This module explains how SQLite fits into this course.

Table of contents

- [Discussion](#)
- [Miscellaneous](#)

Discussion

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

SQL stands for **Structured Query Language** . SQL is used to communicate with a database. According to ANSI (*American National Standards Institute*) , it is the standard language for relational database management systems.

According to [The Python Standard Library - 12.6. SQLite3 - DB-API 2.0 interface for SQLite databases](#),

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language.

SQL is a very broad topic. Considerable time, effort, and experience is required to master SQL. As evidence of this fact, see the somewhat voluminous [W3Schools SQL Tutorial](#).

Knowledge of SQL is a prerequisite for understanding SQLite. It is unreasonable to expect a student to learn SQL during a small portion of a one-semester course in Python.

Take-home assignments requiring the use of SQLite may be based on the material on any of the following websites:

- [W3Schools SQL Tutorial](#)
- [The Python Standard Library - 12.6. SQLite3 - DB-API 2.0 interface for SQLite databases](#)
- [Python Central - Introduction to SQLite in Python](#)
- [udemy - SQLite for Beginners](#)
- [tutorialspoint - SQLite Tutorial](#)
- [ZetCode - SQLite tutorial](#)

However, because knowledge of SQL is a prerequisite for understanding SQLite, and it is unreasonable to expect a student to learn SQL during a small portion of a one-semester course in Python, tests in this course will not include questions about SQL or SQLite. Therefore, this module does not contain review questions regarding SQLite.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1950-SQLite Database
- File: Itse1359-1950.htm
- Published: 11/10/14
- Revised: 12/29/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2110-Preface to Putting Python to Work
Preface to the sub-collection titled Putting Python to Work

Table of contents

- [Discussion](#)
- [Miscellaneous](#)

Discussion

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This material is provided for the benefit of those students who are interested in expanding their learning experience beyond the minimum requirements of the course titled *ITSE 1359 Introduction to Scripting Languages: Python*. Knowledge of the material in this sub-collection titled **Putting Python to Work** is not a requirement of the course.

During my 20+ years of teaching computer programming, I have learned that many students are more likely to succeed in the course if I provide interesting and useful projects for them to work on in addition to simply computing sales-tax tables, etc. Therefore, I try to provide projects such as image processing, network programming, event-driven programming, game programming, etc. in addition to the more boring text-only types of projects in most of the courses that I teach.

Several interesting libraries and Ebooks for doing this sort of programming with Python are freely available on the web including the following:

- [Pygame](#) - special effects, simulations, and games.
- [Blender](#) - 3D modeling
- [Guzdial's book](#) - media computation (images, audio, etc.)
- [Python at Codecademy](#) - an interactive Python tutorial
- [Livewires](#) - computer animation and programming
- [Matplotlib](#) - 2D plotting

- [Python Imaging Library \(PIL\)](#) - image processing
- [Pillow](#) - image processing
- [John Zelle's Graphics Module](#) - graphics and animation
- [Pythonxy](#) - scientific and engineering software for numerical computations
- [Numpy and SciPy](#) - mathematical and numerical routines in pre-compiled, fast functions often required by other libraries.
- [The Invent with Python Bookshelf](#) - A large number of free downloadable Python Ebooks

Many of the items in the above list could be used to develop interesting and useful programming projects for use in this course. There is a problem, however. Many of the items in the list were developed using Python version 2.x and have not yet been updated for compatibility with Python 3.x or later as of January 2015. Therefore, some of the items in the above list are not compatible with the version of Python being used for this course.

As time goes on, I plan to monitor the state of these items and to update the contents of this sub-collection to explore the use of the items in this portion of the course material. Therefore, you should consider this portion of the course material to be a work in process.

I will begin with [Pygame](#), which is compatible with Python version 3.4.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2110-Preface to Putting Python to Work
- File: Itse1359-2110.htm
- Published: 01/02/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2210-Getting Started with Pygame

This module explains how to get started programming with Pygame and Python.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Installing Pygame](#)
- [Discussion and sample code](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

As I mentioned in the earlier module titled [Preface to Putting Python to Work](#) I like to provide my students with interesting and useful programming projects in addition to the dull and boring text-based projects typically found in beginning programming courses. This is the first module in a sub-collection that will explore the use of the [Pygame](#) library for writing programs involving drawing, color, fonts, graphics, bitmap images, sound, animation, music, and sprites, along with other assorted and interesting effects.

This material is provided for the benefit of those students who are interested in expanding their learning experience beyond the minimum requirements of the course titled *ITSE 1359 Introduction to Scripting Languages: Python*

. Knowledge of the material in this sub-collection titled **Pygame** is not a requirement of the course.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Graphic screen output.

Listings

- [Listing 1.](#) Source code for TestProgram.py.

General background information

The Pygame home page is located at <http://www.pygame.org>. As of January 2015, if you open that page you will find a menu on the upper-left that provides links to the other pages on the website such as [Downloads](#), [Documentation](#) and [Tutorials](#).

If you select [About](#) in that menu, you will find a description of Pygame that reads partially as follows:

Note: Pygame

Pygame is a set of Python modules designed for writing games. Pygame adds functionality on top of the excellent [SDL](#) library. This allows you to create fully featured games and multimedia programs in the python

language. Pygame is highly portable and runs on nearly every platform and operating system...

Pygame is free. Released under the [GPL License](#), you can create open source, free, freeware, shareware, and commercial games with it. See the license for full details.

For a nice introduction to pygame, examine the [Line By Line Chimp tutorial](#)... or [Chapters 17 to 20](#) of [Invent with Python](#).

Installing Pygame

Because of the different versions of Python and Pygame that are available, and because of the different operating systems in use, selection of the correct set of files to download and install can be complicated.

Unfortunately, as of January 2, 2015, the **Install** link at <http://www.pygame.org/docs/> is broken. This link purports to provide

"Steps needed to compile Pygame on several platforms. Also help on finding and installing prebuilt binaries for your system."

Fortunately, the information is available from other sources on the web.

If you are currently using or are willing to use Python v3.4 on a Windows machine, in his chapter titled *"Before getting started..."* [Professor Paul Vincent Craven](#) recommends simply that you

1. First run the Python installer downloaded from:
ProgramArcadeGames.com/python-3.4.2.msi
2. Then run the Pygame installer downloaded from:
ProgramArcadeGames.com/pygame-1.9.2a0.win32-py3.4.msi

In my case, I already had Python v3.4 installed on my Windows 7 machine. I downloaded and ran the Pygame installer listed above. As I recall, the installer asked if I wanted to install the program based on the location of

Python 3.4 or elsewhere. I elected to install it based on the location of Python 3.4. This resulted in the Pygame library being integrated into the Python34 directory structure on my C-drive in such a way that the same *path environment variable* that I have been using since the beginning of the course works for Pygame programs.

If you are using an operating system other than Windows, or if you are using Windows and are using a version of Python other than v3.4, Al Sweigart provides detailed installation instructions in [Chapter 17](#) of [Invent with Python](#).

Discussion and sample code

I am providing the Pygame program shown in [Listing 1](#) in this module solely for test purposes with no explanation. I will explain it, or something similar, in a future module.

Listing 1. Source code for TestProgram.py.

```
"""
File TestProgram.py
Draws a transparent rectangle with an opaque
red border on
a green background.
"""

# Import pygame library and initialize all
imported pygame modules
import pygame
pygame.init()
```

Listing 1. Source code for TestProgram.py.

```
#Initialize color constants
GREEN    = ( 0, 255, 0)
RED      = ( 255, 0, 0)

#Initialize the display Surface object
displaySurface =
pygame.display.set_mode([250,150])

#Set the window caption
pygame.display.set_caption("TestProgram")

#Initialize control variable used for
termination by the user.
quit = False

#Create an object to help track time
timer = pygame.time.Clock()

#Execute the runtime loop
while not quit:

    #Get user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Final iteration of
runtime loop.

    #Fill display Surface object with solid
green
    displaySurface.fill(GREEN)

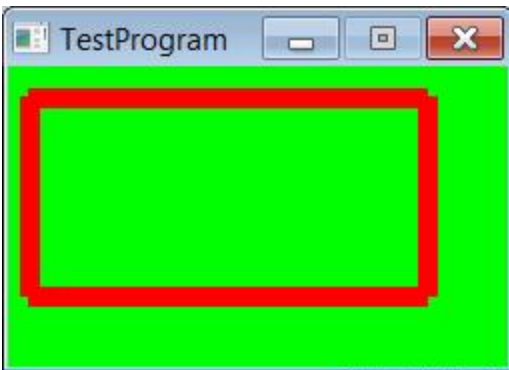
    #Draw a rectangle on the display Surface
object
    pygame.draw.rect(displaySurface, RED, [10,
15, 200, 100], 10)
```

Listing 1. Source code for TestProgram.py.

```
#Copy contents of display Surface object  
to the physical  
# screen.  
pygame.display.flip()  
  
#Control the frame rate by implementing a  
delay that does not  
# use much cpu resource.  
timer.tick(20)  
  
#Outside the runtime loop  
pygame.quit()
```

If you have Python and Pygame properly installed on your computer, you should be able to copy and run the code shown in [Listing 1](#) to produce the screen output shown in [Figure 1](#).

Figure 1. Graphic screen output.



Run the program

I encourage you to copy the code from [Listing 1](#). Execute the code and confirm that you get the graphic screen output shown in [Figure 1](#). Experiment with the code, making changes, and observing the results of

your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2210-Getting Started
- File: Itse1359-2210.htm
- Published: 01/02/15
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2215-Structure of a Pygame Program

This module explains the polling pattern often used for game or simulation programs.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [The simple fall through pattern](#)
 - [The event-driven pattern](#)
 - [The polling pattern](#)
- [Discussion and sample code](#)
 - [Import required libraries](#)
 - [Initialize everything that needs to be initialized](#)
 - [Initialize imported pygame modules](#)
 - [Initialize color constants](#)
 - [Initialize control variable used for termination by the user](#)
 - [Initialize control variable used to determine colors of background and border](#)
 - [Set the window caption](#)
 - [Create objects that will be used by the program](#)
 - [Create the display Surface object](#)
 - [Create an object to help track time](#)
 - [Enter the runtime loop and continue looping until a "quit" signal is received](#)

- [Get user inputs for this iteration of the runtime loop](#)
- [Get state of internal control structures for this iteration of the runtime loop](#)
- [Execute the program logic based on the state of internal control structures](#)
 - [Set background and border colors on the basis of the random number](#)
 - [Fill display_Surface object with backgroundColor](#)
 - [Draw a rectangle on the display_Surface object with borderColor](#)
- [Display the new state of the program](#)
- [Deal with frame rate control](#)
- [Do any required cleanup and terminate the program](#)
- [Other observations](#)
 - [The location of the origin](#)
 - [The minimize, maximize, and quit buttons](#)
 - [The appearance of the display window](#)
 - [A partially covered display window](#)
 - [Uncovering the display window](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Pseudo code for a polling framework.
- [Figure 2.](#) Typical operations inside the runtime loop.
- [Figure 3.](#) Program output for one state.
- [Figure 4.](#) Program output for a different state.
- [Figure 5.](#) The origin of the display window.
- [Figure 6.](#) The appearance of the display window.
- [Figure 7.](#) A partially covered display window.
- [Figure 8.](#) Uncovering the display window.

Listings

- [Listing 1.](#) Import required libraries.
- [Listing 2.](#) Initialize everything that needs to be initialized.
- [Listing 3.](#) Create objects that will be used by the program.
- [Listing 4.](#) Enter the runtime loop and continue looping until a "quit" signal is received.
- [Listing 5.](#) Get user inputs for this iteration of the runtime loop.
- [Listing 6.](#) Get state of internal control structures for this iteration of the runtime loop.
- [Listing 7.](#) Execute the program logic based on the state of internal control structures.
- [Listing 8.](#) Display the new state of the program.
- [Listing 9.](#) Deal with frame rate control by implementing a delay that does not use much cpu resource.
- [Listing 10.](#) Having exited the runtime loop, do any required cleanup and terminate the program.
- [Listing 11.](#) Complete program listing.

General background information

The structure of a computer program usually takes on one of several familiar patterns. There are many patterns and many variations on each pattern. Three common patterns are described below.

The simple fall through pattern

One such pattern is the "*simple fall through*" pattern often used for teaching programming to beginning programming students. With this pattern, the program starts, does something, and then terminates. In other words, it "*falls through*" the code performing some required action (*such as printing a tax table*) along the way.

The event-driven pattern

This is the pattern commonly used for programs such as spread sheets, word processors, inventory control programs, and other programs that are expected to run for long periods of time with frequent periods during which the program is idle. This type of program lends itself to having many programs running concurrently on a single computer because it consumes minimal cpu resources when idle.

This type of program typically start up, does some initialization, and then goes to sleep waiting for the user to create an event such as a *key press* , a *mouse click* , etc. When an event happens, the program wakes up, services the event, and then goes back to sleep. As mentioned above, while sleeping, the program consumes very little in the way of cpu resources allowing those resources to be used by other programs.

Overall program control during idle periods is handled behind the scenes and out of sight of the programmer. In particular, the programmer doesn't write code to determine if an event has happened. Instead, the program is automatically notified when an event happens and the programmer writes

the code required to service the event. If the programmer isn't interested in a particular type of event, a service method is not written for that event type and it is ignored with no further effort on the part of the programmer.

The polling pattern

This pattern is often used for programs that are intended to be very active from start to finish such as games and simulations. Programs of this sort may not be intended to *"play nicely and share cpu resources"* with other programs running concurrently on the same computer.

With this pattern, it is the responsibility of the programmer to "poll" the state of input devices and internal control structures to determine if something interesting has happened and then to take appropriate action when something interesting does happen. It is also the responsibility of the programmer to exercise some control over the amount of cpu resources consumed by the program. The programmer often has the ability to starve other programs of cpu resources in order to improve the performance of his or her program.

Pygame programs generally fall in this category. Also, programs written with the XNA Game Studio, Allegro, Dark GDK, Slick2D, and many other frameworks also fall in this category.

Although there are many variations and many degrees of sophistication among the frameworks, the overall structure of such frameworks typically looks something like the pseudo code shown in [Figure 1](#).

Figure 1. Pseudo code for a polling framework.

Figure 1. Pseudo code for a polling framework.

- Start
- Import required libraries.
- Load resources such as sound and image files.
- Define classes, methods, and functions that will be used by the program.
- Initialize everything that needs to be initialized.
- Create objects that will be used by the program.
- Enter the runtime loop and continue looping until a "quit" signal is received.
 - Get user inputs for this iteration of the runtime loop.
 - Get states of internal control structures for this iteration of the runtime loop.
 - Execute the program logic based on the input values and the state of internal control structures.
 - Display the new state of the program.
 - Deal with frame rate control if required by the framework.
Then either start a new iteration or exit the runtime loop.
- Having exited the runtime loop, do any required cleanup and terminate the program - Quit.

The preliminary operations prior to entering the runtime loop vary widely from one program to the next. However, most game and simulation programs will implement the five operations shown in [Figure 2](#) (*and possibly other operations as well*) inside the runtime loop.

Figure 2. Typical operations inside the runtime loop.

1. Get user inputs for this iteration of the runtime loop.
2. Get states of internal control structures for this iteration of the runtime loop.
3. Execute the program logic based on the input values and the state of internal control structures.
4. Display the new state of the program.
5. Deal with frame rate control if required by the framework. Then either start a new iteration or exit the runtime loop.

Discussion and sample code

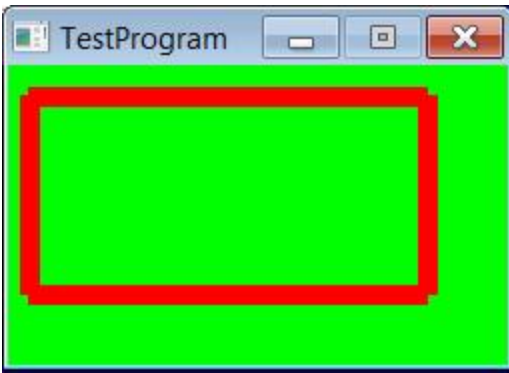
The code in [Listing 11](#) provides a simple illustration of the five operations listed in [Figure 2](#). The comments in the program generally coincide with the operations listed in [Figure 2](#).

The program produces the outputs shown in [Figure 3](#) and [Figure 4](#) at different points in time.

Figure 3 . Program output for one state.



Figure 4. Program output for a different state.



As shown in [Figure 3](#) and [Figure 4](#), the output switches between a rectangle with a red border on a green background and a rectangle with a green border on a red background. The switch occurs randomly on the basis of a random number and can occur as frequently as once per second.

I will break this program down and explain it in fragments.

Import required libraries

[Listing 1](#) shows that portion of the program that imports the required libraries.

Listing 1. Import required libraries.

```
import random
import pygame
```

The **random** module is imported for the purpose of generating a random sequence of 0 and 1 values. These values are used to determine the color of

the border and the background that will be displayed during each iteration of the runtime loop.

The **python** library is imported to provide a variety of capabilities to the program.

Initialize everything that needs to be initialized

The code in [Listing 2](#) initializes everything that needs to be initialized.

Listing 2. Initialize everything that needs to be initialized.

```
#Initialize imported pygame modules
pygame.init()

#Initialize color constants
GREEN    = ( 0, 255, 0)
RED      = ( 255, 0, 0)

#Initialize control variable used for
termination by the user.
quit = False

#Initialize control variable used to determine
colors of background and border.
randomNumber = 0;

#Set the window caption
pygame.display.set_caption("TestProgram")
```

Initialize imported pygame modules

[Listing 2](#) begins by initializing the imported **pygame** modules. This statement, or something similar is required for all programs that use the **pygame** library.

Initialize color constants

After that, the code in [Listing 2](#) initializes the values for two constants that represent the colors GREEN and RED. I will explain more about how **pygame** deals with colors in a future module.

Initialize control variable used for termination by the user

The variable named **quit** is initialized to **False** . Later on, you will see how it is changed to **True** to cause the runtime loop and hence the program to terminate.

Initialize control variable used to determine colors of background and border

The variable named **randomNumber** is initialized to a value of zero. Later on, it will receive a new value of either 0 or 1 from a random number generator once during each iteration of the runtime loop. You will see later how this value is used to control the colors of the border and background in [Figure 3](#) and [Figure 4](#).

Set the window caption

Finally, the code in [Listing 2](#) sets the caption in the banner at the top of the window that appears in [Figure 3](#) and [Figure 4](#).

Create objects that will be used by the program

The code in [Listing 3](#) creates the objects that will be used later by the program.

Listing 3. Create objects that will be used by the program.

```
#Create the display Surface object
displaySurface =
pygame.display.set_mode([250,150])

#Create an object to help track time
timer = pygame.time.Clock()
```

Create the display Surface object

[Listing 3](#) begins by creating and setting the mode for an object of type **Surface**. In this case, the mode is a rectangular window 250 pixels wide and 150 pixels tall as shown in [Figure 3](#). (*I will have more to say about the **set_mode** method [later](#).*)

A **Surface** object is essentially an area of memory that represents a portion of the screen. Program code can draw on the surface without the drawing being visible on the screen. Later, a method named **flip** can be called to copy the drawing from the surface to the screen very quickly. This provides a more pleasing effect than drawing directly on the screen, particularly in those cases where a noticeable amount of time is required to complete the drawing.

Create an object to help track time

The object referred to by the variable named **timer** in [Listing 3](#) will be used later to insert a one-second delay at the end of each iteration of the runtime loop. A delay implemented in this manner consumes very little in the way of cpu resources and prevents the computer from starving other programs of cpu resources that they might need to complete their tasks.

Enter the runtime loop and continue looping until a "quit" signal is received

The code in [Listing 4](#) shows the beginning of a **while** loop that I will refer to as the runtime loop.

Listing 4. Enter the runtime loop and continue looping until a "quit" signal is received.

```
while not quit:
```

This loop will continue to iterate until the value of the variable named **quit** (see [Listing 2](#)) changes from **False** to **True** . Usually the bulk of the time encompassed by a computer simulation or computer game involves millions of iterations of the runtime loop.

Get user inputs for this iteration of the runtime loop

This is where the term [polling](#) comes into play. For programs written using the [polling.pattern](#), the programmer must write code to *"poll the system"* to

determine what, if any user input has occurred. That polling is illustrated by the code in [Listing 5](#).

Listing 5. Get user inputs for this iteration of the runtime loop.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        quit = True #Final iteration of
runtime loop.
```

The user can provide input to the program in several different ways including:

- QUIT
- KEYDOWN
- KEYUP
- MOUSEMOTION
- MOUSEBUTTONUP
- MOUSEBUTTONDOWN

However, unlike with the [event-driven pattern](#), the program will only learn about those user actions that it inquires about in the polling code. In this program, for example, the only action that the user can take that will become known to the program is to click the red x-button in the upper-right of [Figure 3](#). When that happens, an event type matching **pygame.QUIT** occurs. This event type is recognized by the **if** statement in [Listing 5](#). This in turn causes the value of the variable named **quit** to be switched from the initial value of **False** to a new value of **True**, which in turn causes the runtime loop to terminate at the end of the current iteration. For a real game

or simulation, the code in [Listing 5](#) would probably be expanded considerably to make it possible for other user actions to be recognized by the program.

A big difference between the event-driven pattern and the polling pattern has to do with when the program learns of the occurrence of the event. With the polling pattern, the program will only learn of the event at the beginning of the next iteration of the runtime loop, which could involve a noticeable delay. With the event-driven pattern, the program is usually notified of the event very quickly after it occurs and may or may not be able to service the event very quickly depending on other circumstances.

Get state of internal control structures for this iteration of the runtime loop

Other than the QUIT button discussed above, the only internal control structure in this program that is used to control the behavior of the program is the variable named **randomNumber** . As you will see later, the value of that variable determines the colors of the border and the background in [Figure 3](#) and [Figure 4](#).

Listing 6. Get state of internal control structures for this iteration of the runtime loop.

```
randomNumber = random.randrange(2)
```

The code in [Listing 6](#) calls a random number generator to store either a 0 or a 1 in that variable. That value constitutes the current state of the control variable and in this program constitutes the current state of the control

structure. A real simulation or game program would typically have many different control variables and the combination of the individual states of those control variables would define the state of the overall control structure.

Execute the program logic based on the state of internal control structures

The code in [Listing 7](#) executes the program logic based on the state of internal control structures. Other than the termination of the runtime loop when the user clicks the x-button, there is no logic in this program based on user input.

Listing 7. Execute the program logic based on the state of internal control structures.

Listing 7. Execute the program logic based on the state of internal control structures.

```
#Set background and border colors on the
basis of the random number.
if randomNumber == 0:
    borderColor = GREEN
    backgroundColor = RED
else:
    borderColor = RED
    backgroundColor = GREEN

#Fill display Surface object with
backgroundColor
displaySurface.fill(backgroundColor)

#Draw a rectangle on the display Surface
object with borderColor.
pygame.draw.rect(displaySurface,
borderColor, [10, 15, 200, 100], 10)
```

Set background and border colors on the basis of the random number

[Listing 7](#) begins by setting the background and border colors on the basis of the random number using the color constants that were initialized in [Listing 2](#).

Fill display Surface object with backgroundColor

Then [Listing 7](#) calls the **fill** method on the **Surface** object to, using the terminology of the [documentation](#), *"fill Surface with a solid color."* In other

words, the code paints the entire surface with the color specified by the value currently stored in the variable named **backgroundColor** .

Draw a rectangle on the display Surface object with `borderColor`

Finally the code in [Listing 7](#) draws a rectangle on the display surface. The color of the border is specified as the value contained in the variable named **borderColor** . The width of the border is given by the last parameter value, which is 10 pixels.

The upper-left corner of the rectangle is located at (10,15). The rectangle is 200 pixels wide and 100 pixels tall as shown in [Figure 3](#).

Note that the coordinate values that specify the position of the upper-left corner of the rectangle are relative to the upper-left corner of the red or green background area of the display window, not relative to the upper-left corner of the screen. Also note that half the specified border width is outside the actual border and half is inside. Therefore, specifying a border width greater than one pixel actually increases the overall width and height of the rectangle.

Display the new state of the program

The code in [Listing 8](#) calls the **flip** method to very quickly copy the drawing that has been produced on the **Surface** object to the screen.

Listing 8. Display the new state of the program.

Listing 8. Display the new state of the program.

```
pygame.display.flip()
```

Because this method is called inside of the runtime loop, the image on the screen will be refreshed once during each iteration of the runtime loop. One effect of this is that if a portion of the output is covered and then uncovered by another window, it will be redrawn shortly after being uncovered.

Because this program is refreshing the image on the screen only once per second, if you cover and then uncover it with another window, you will probably see a delay before the uncovered portion is redrawn on the screen. You may also notice a delay of up to one second when you click the x-button to terminate the program. I will discuss this in more detail later in the section titled [Other observations](#).

Deal with frame rate control

[Listing 9](#) calls the **tick** method of the **Clock** object to deal with frame rate control by implementing a delay that does not use much cpu resource.

Listing 9. Deal with frame rate control by implementing a delay that does not use much cpu resource.

```
timer.tick(1)
```

The [documentation](#) describes the **tick** method as shown follows:

Note: The tick method of the Clock object

This method should be called once per frame. It will compute how many milliseconds have passed since the previous call.

If you pass the optional framerate argument the function will delay to keep the game running slower than the given ticks per second. This can be used to help limit the runtime speed of a game. By calling `Clock.tick(40)` once per frame, the program will never run at more than 40 frames per second. Note that this function uses `SDL_Delay` function which is not accurate on every platform, but does not use much cpu. Use `tick_busy_loop` if you want an accurate timer, and don't mind chewing cpu.

Do any required cleanup and terminate the program

Having exited the runtime loop, [Listing 10](#) calls `pygame.quit` to *"Uninitialize all pygame modules that have previously been initialized."*

Listing 10. Having exited the runtime loop, do any required cleanup and terminate the program.

```
pygame.quit()
```

There is some disagreement as to the need to call this function. The [documentation](#) states *"When the Python interpreter shuts down, this method is called regardless, so your program should not need it, except when it wants to terminate its pygame resources and continue."*

On the other hand [Professor Paul Vincent Craven](#) tells us that it is needed to resolve a deficiency in the IDLE development environment.

In any event, the [documentation](#) also states *"It is safe to call this function more than once: repeated calls have no effect."* Therefore, I will take Professor Craven's advice and call the function at the end of each of my **pygame** programs.

Other observations

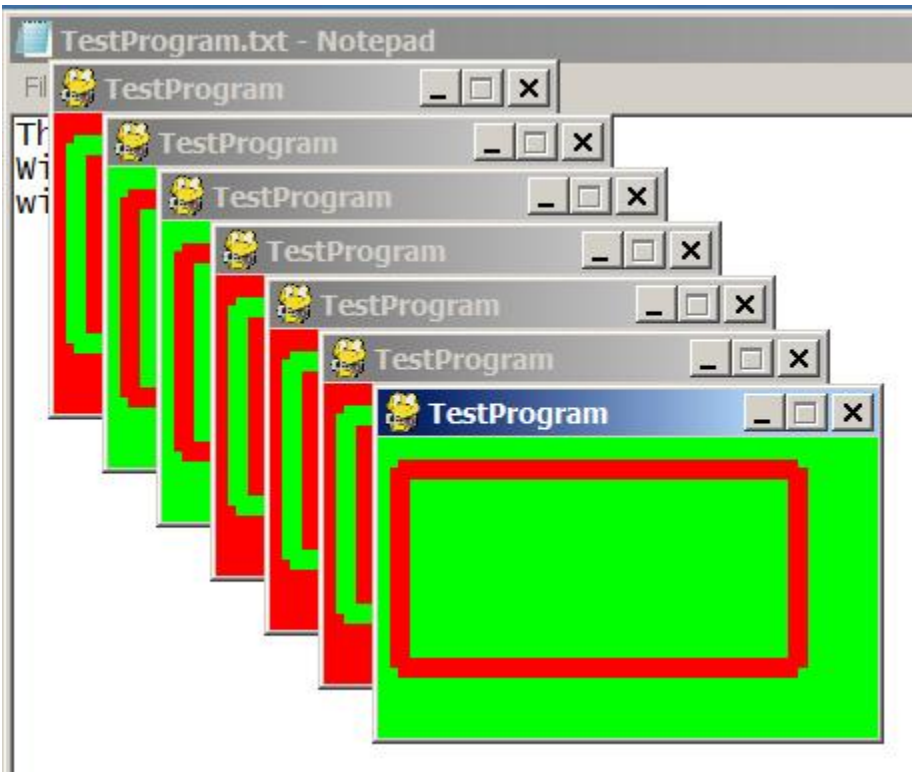
According to the [documentation](#), *"Pygame has a single display Surface that is either contained in a window or runs full screen."* In other words, it is apparently not possible to write a program using **pygame** that places multiple display windows on the screen.

Some aspects of display window are controlled by code that you write but many aspects are beyond the direct control of the programmer.

The location of the origin

For example, according to the [documentation](#), *"The origin of the display, where $x = 0$, and $y = 0$ is the top left of the screen"* On my Windows 7 machine, the origin is as shown by the **pygame** display windows in [Figure 5](#). *(The notepad window was manually placed in the top left corner of the screen before running the **pygame** program multiple times.)*

Figure 5. The origin of the display window.



When running from the command line, you can run multiple instances of the **pygame** program. (*Although not demonstrated here, it appears that the state of each instance is independent of the state of all other instances.*) The origin of the display window for the first instance is near but not at the top left corner of the screen. The origin for each of the next six instances is placed to the right of and slightly below the origin for the previous instance as shown in [Figure 5](#). The origin for the eighth instance is the same as the origin for the first instance and the pattern repeats.

Although you cannot display multiple instances when running from the IDLE IDE or the Wing IDE, similar behavior regarding placement occurs if you run, kill, and then rerun the program multiple times from within the IDE.

The display window can be moved manually once it appears on the screen. However, it is not resizable by default. It can be made resizable by passing the **pygame.RESIZABLE** flag to the **set_mode** method in the code that you write.

The minimize, maximize, and quit buttons

As is typical under the Windows operating system, the display window has three buttons in the upper-right corner as shown in [Figure 5](#):

- minimize - *(the one on the left with the underscore character)*
- maximize - *(the one in the middle with the rectangle that appears to be disabled)*
- quit - *(the one on the right with the X)*

The display window will respond to the **minimize** button and the **quit** button without any requirement for you to write code to support that action. However, it will not respond to the **maximize** button.

Clicking the **minimize** button will cause the display window to be minimized into an icon in the system tray. Clicking the icon in the system tray will restore the window to its previous location on the screen.

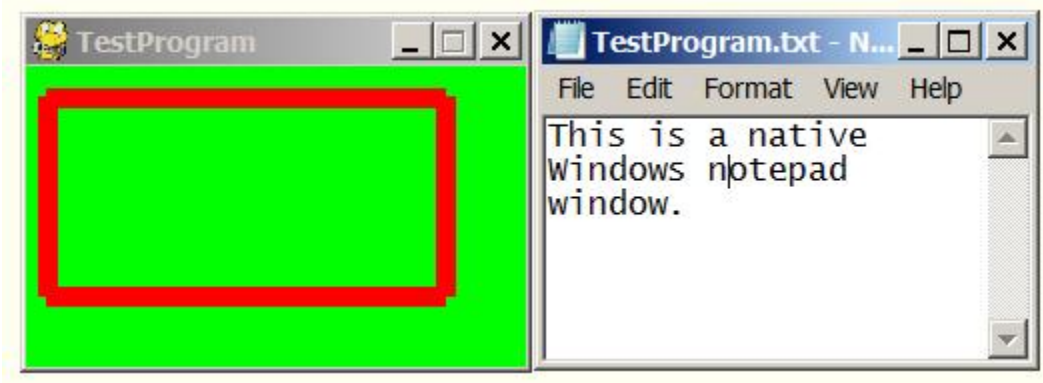
The documentation is somewhat confusing regarding the **quit** button but the Windows Task Manager can be used to confirm that clicking the **quit** button will terminate the program.

As an alternative to the maximize button, you can cause the program to run in full screen mode by passing the **pygame.FULLSCREEN** flag to the **set_mode** method. *(Be careful with this one. If not used properly, you may have to shut down your computer to get out of full screen mode.)*

The appearance of the display window

You may find that the **pygame** display window looks like the native window on your operating system. [Figure 6](#) shows a pygame display window alongside a notepad window on Windows 7 with the Windows appearance option set to *Windows Classic*. As you can see, they are very similar. *(Note that the appearance in [Figure 6](#) is somewhat different from the appearance in [Figure 4](#), which was not run under Windows Classic.)*

Figure 6. The appearance of the display window.

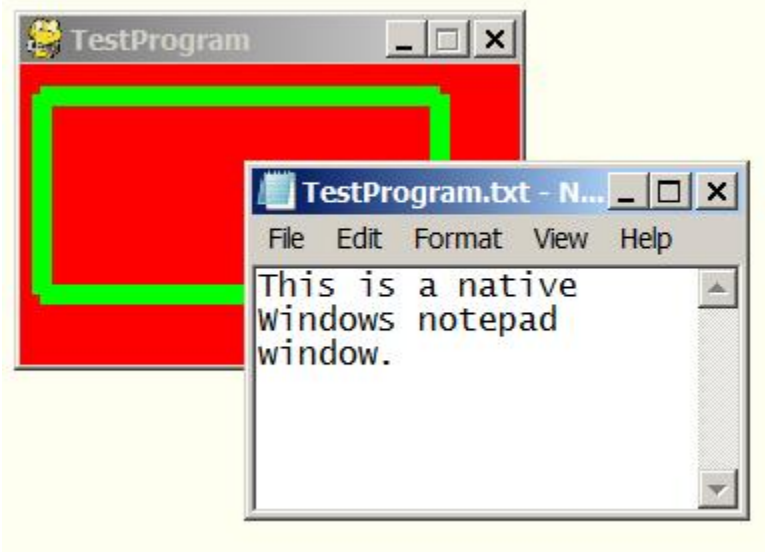


Windows has many appearance options. Switching between appearance options causes changes to the appearance of the **pygame** display window (*slightly rounded corners for example as shown in [Figure 4](#)*) but does not cause the appearance of the **pygame** display window to track the appearance of a Windows notepad window in all respects. For whatever its worth, however, selection of the *Dell Aero Theme* on my Dell computer running Windows 7 causes the **pygame** display window and the notepad window to look almost identical including button size, rounded corners, backshadow, border width, and border transparency.

A partially covered display window

[Figure 7](#) shows what happens when another window partially covers the **pygame** display window on a Windows 7 system.

Figure 7. A partially covered display window.



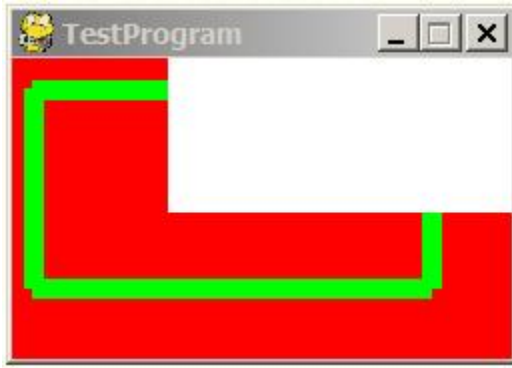
The result is that the **pygame** program continues to run and to display changing graphics in the visible portion of the display window. This is true even though the **pygame** program no longer *"has the focus."* Among other things, this probably means that the program logic will continue to keep the state of the game or simulation current when the program loses focus provided that it has access to sufficient cpu resources. If the **pygame** program in [Figure 7](#) regains the focus, the display window will cover the notepad window.

Uncovering the display window

[Figure 8](#) shows what you are likely to see if you

- Slow the frame rate down to perhaps one iteration of the runtime loop every ten seconds.
- Partially cover the display window with another window.
- Uncover the display window.

Figure 8. Uncovering the display window.



The white rectangle in [Figure 8](#) is the portion of the display window that was covered. (*Note that the banner at the top of the window including the buttons was also partially covered with the other window.*) The white area will continue to be white until the next iteration of the runtime loop in which the code calls the **flip** method on the **Surface** object.

The image in [Figure 8](#) illustrates a couple of important points.

- First, the display of the window in terms of the banner, buttons, border, etc., is independent of the frame rate of the program. Those portions of the window are restored as soon as the display window is uncovered.
- Second, the code that you write only controls the graphics contents of the window in the area below the banner and inside the borders. This area is often referred to as the "*client area*" of a window.

Although not shown here, you can eliminate all but the client area of the display window by passing the **pygame.NOFRAME** flag to the **set_mode** method. If you do that and run the same experiment, the white area will appear and will remain on the screen until the next iteration of the runtime loop, at which time the normal graphics will be displayed. (*Make sure you have a way to terminate the program before doing that.*)

Run the program

I encourage you to copy the code from [Listing 11](#). Execute the code and confirm that you get the same results as those shown in [Figure 3](#) through [Figure 8](#). Experiment with the code, making changes, and observing the

results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program discussed in this module is provided below.

Listing 11. Complete program listing.

```
"""
File TestProgram.py
Revised: 01/02/15
The output switches between a rectangle with a red
border on a green background
and a rectangle with a green border on a red
background.
```

```
The switch occurs randomly on the basis of a
random number and can occur as
frequently as once per second.
```

```
=====
=====
```

```
"""
#===Import required libraries.===
import random
import pygame
```

```
#===Initialize everything that needs to be
initialized.===
#Initialize imported pygame modules
pygame.init()
```

```
#Initialize color constants
GREEN    = ( 0, 255, 0)
RED      = ( 255, 0, 0)
```

```
#Initialize control variable used for termination
by the user.
quit = False
```

```
#Initialize control variable used to determine
colors of background and border.
randomNumber = 0;
```

```
#Set the window caption
pygame.display.set_caption("TestProgram")
```

```
#####Create objects that will be used by the
program.###
#Create the display Surface object
displaySurface =
pygame.display.set_mode([250,150])
```

```
#Create an object to help track time
timer = pygame.time.Clock()
```

```
#####Enter the runtime loop and continue looping
until a "quit" signal is
#   received.###
```

```
while not quit:
    #####Get user inputs for this iteration of the
runtime loop.###
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Final iteration of
runtime loop.
```

```
    #####Get state of internal control structures
for this iteration of the
#   runtime loop.###
```

```

    randomNumber = random.randrange(2)

    #===Execute the program logic based on the
input values and the state of
    #    internal control structures.===

    #Set background and border colors on the basis
of the random number.
    if randomNumber == 0:
        borderColor = GREEN
        backgroundColor = RED
    else:
        borderColor = RED
        backgroundColor = GREEN

    #Fill display Surface object with
backgroundColor
    displaySurface.fill(backgroundColor)

    #Draw a rectangle on the display Surface
object with borderColor.
    pygame.draw.rect(displaySurface, borderColor,
[10, 15, 200, 100], 10)

    #===Display the new state of the program.===
    pygame.display.flip()

    #===Deal with frame rate control by
implementing a delay that does not
    # use much cpu resource.===
    timer.tick(1)

    #===Having exited the runtime loop, do any
required cleanup and terminate the
    #    program.
    pygame.quit()

```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2215-Structure of a Pygame Program
- File: Itse1359-2215.htm
- Published: 01/31/16
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [The graphic output](#)
 - [The program code](#)
 - [Import and initialize pygame library.](#)
 - [Perform some initializations](#)
 - [Create some objects](#)
 - [Overloaded constructors, methods, and functions](#)
 - [An object of type Rect](#)
 - [Two objects of type Color](#)
 - [Create and set display mode on Surface object](#)
 - [An object of type Clock](#)
 - [Enter the runtime loop](#)
 - [Get and process user inputs](#)
 - [Fill display Surface object with green background Color object](#)
 - [Cause border Color object to morph from red to green](#)
 - [Draw and display the rectangle](#)
 - [Terminate the program outside the runtime loop](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame**.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Output near beginning of cycle.
- [Figure 2.](#) Output near middle of cycle.
- [Figure 3.](#) Output near end of cycle.

Listings

- [Listing 1.](#) Import and initialize pygame library.
- [Listing 2.](#) Perform some initializations.
- [Listing 3.](#) Create some objects.
- [Listing 4.](#) Enter the runtime loop.
- [Listing 5.](#) Get and process user inputs.
- [Listing 6.](#) Fill display Surface object with green background Color object.
- [Listing 7.](#) Cause border Color object to morph from red to green.
- [Listing 8.](#) Draw and display the rectangle.
- [Listing 9.](#) Terminate the program outside the runtime loop.
- [Listing 10.](#) Complete program listing.

General background information

Just about any program that you write that is intended for human consumption will involve color. Therefore, you need to understand how Python and Pygame handle color.

Color is a very broad topic so this won't be the final module on this topic. There are at least six different ways that color can be represented in a **pygame** program:

- RGB and RGBA
- CMY
- HSV and HSVA
- HSL and HSLA
- |1|2|3
- #rrggbb and #rrggbbaa

RGB and RGBA are the most fundamental representations. The letters in the acronym represent the following color components (*often called channels*) :

- R - red
- G - green
- B - blue
- A - alpha (*transparency*)

Alpha transparency is not required for an understanding of the basic RGB color system so a discussion of alpha will be deferred until a future module. This module will concentrate on the RGB representation of color along with objects of the **Color** and **Rect** classes.

Discussion and sample code

I will explain a program that draws a rectangle with a wide colored border on a green background. The border on the rectangle starts out as red. Over time, it morphs into green. When it reaches green, making it indistinguishable from the background, it resets to red and the cycle continues.

The graphic output

[Figure 1.](#), [Figure 2.](#), and [Figure 3.](#) show the screen output at three stages of the cycle.

Figure 1. Output near beginning of cycle.

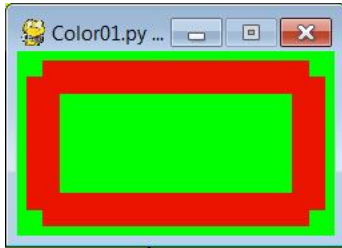
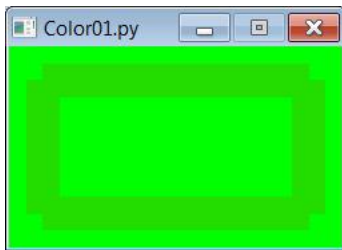


Figure 2. Output near middle of cycle.



Figure 3. Output near end of cycle.



[Figure 1](#) shows the output near the beginning of the cycle. [Figure 2](#) shows the output near the middle of the cycle, and [Figure 3](#) shows the output near the end of the cycle. Shortly after the output shown in [Figure 3](#), the border on the rectangle was reset to pure red.

The program code

A complete listing of the program is provided in [Listing 10](#). I will break the program down and explain it in fragments.

Import and initialize pygame library

As is always the case when programming with **pygame**, [Listing 1](#) imports and initializes the **pygame** library.

Listing 1. Import and initialize pygame library.

Listing 1. Import and initialize pygame library.

```
import pygame #Import required library
pygame.init() #Initialize imported pygame modules
```

Perform some initializations

The code in [Listing 2](#) performs some initializations.

Listing 2. Perform some initializations.

```
quit = False #Initialize termination control variable.
pygame.display.set_caption("Color01.py") #Set the window caption
INCREMENT = 20 #Initialize control constant
WIDTH = 25 #Initialize border width constant
FRAMERATE = 5 #Used to control the maximum frame rate.
```

The variable named **quit** is used to cause the runtime loop to terminate when the user clicks the red X-button in the upper-right corner of the image in [Figure 1](#).

The second statement in [Listing 2](#) sets the caption in the banner at the top of the display window as shown in [Figure 1](#).

The constant named **INCREMENT** will be used later to control how rapidly the border color morphs from red to green.

The constant named **WIDTH** will be used later to specify the width of the rectangle's border.

The constant named **FRAMERATE** will be used later to limit the program to no more than 5 frames per second.

Create some objects

The code in [Listing 3](#) creates some objects that will be used later.

Listing 3. Create some objects.

```
rect = pygame.Rect(25, 25, 200, 100) #create a Rect object
border = pygame.Color(255,0,0) #Create a red Color object
background = pygame.Color( 0, 255, 0) #Create a green Color object
surface = pygame.display.set_mode([250,150]) #Set the display mode
clock = pygame.time.Clock() #Create an object to help track time
```

Overloaded constructors, methods, and functions

Prior to the advent of object-oriented programming (*OOP*) , when programming in Pascal or C for example, if you needed two or more functions in the same scope that would accept the same data in different formats and perform essentially the same operations on that data, you were required to define two or more different functions with different names. This resulted in code that could be confusing due to the use of different function names to perform essentially the same operations.

With OOP, we are allowed to define two or more different functions with the same name in the same scope provided that each function has a different formal argument list. This is known as *function overloading* .

For example, you might want to make it possible to process telephone numbers in either of the following formats:

- ("512","123-4567")
- ("512.123.4567")

Prior to OOP you would have been required to write two different functions with different names. With OOP and function overloading, you can write two different functions with the same name. The programmer who calls the function is only required to remember a single name. The decision as to which overloaded function to execute at runtime is based on the number and types of parameters that are passed to the function. This results in cleaner, more self-documenting code.

Overloading in Python also applies to class constructors and methods as well as to functions. This is one aspect of a concept known as *polymorphism* . In general, polymorphism is a word that means something like "*one name, many forms.*"

Most and perhaps all of the methods and functions in **pygame** that require color information as input are overloaded so as to accept a tuple, a list, or a reference to an object of type **Color** as an input parameter. Similarly, most and perhaps all of the functions that require the numeric description of a rectangle as input will accept a tuple, a list, or a reference to an object of type **Rect** as an input parameter.

One of my objectives in publishing this sub-collection of modules on **pygame** is to help you wrap your mind around the Python/pygame flavor of object-oriented programming. Pygame programs are more self-documenting and more obviously object-oriented when written using objects created from specific classes (*such as **Color** and **Rect***) than when using tuples, lists, and other non-specific constructs. Therefore, the sample programs that I provide in this sub-collection of modules on **pygame** will tend toward the use of objects created from specific classes where possible.

An object of type Rect

According to the [documentation](#), the **Rect** class is specifically designed to store and manipulate rectangular areas. Once an object of type **Rect** is constructed, the object provides numerous methods and attributes that can be used to manipulate the rectangular area such as **move** , **inflate** , **clip** , **union** , **contains** , **colliderect** , etc.

One of the overloaded constructors for an object of type **Rect** accepts numeric parameters for **left** , **top** , **width** , and **height** as incoming parameters. Other overloaded constructors are defined with different formal argument lists.

The **left** and **top** parameters specify the coordinates of the upper-left corner of the rectangle in pixels. The **width** and **height** parameters specify the width and the height of the rectangle in pixels.

The first statement in [Listing 3](#) creates a new object of type **Rect** with the numeric parameters shown and saves a reference to that object in the variable named **rect** . This reference will be passed as a parameter to the **pygame.draw.rect** method later to cause a rectangle to be drawn on the display surface.

Two objects of type Color

According to the [documentation](#), *"The Color class represents RGBA color values using a value range of 0-255. It allows basic arithmetic operations to create new colors, supports conversions to other color spaces such as HSV or HSL and lets you adjust single color channels. Alpha defaults to 255 when not given."*

As with the **Rect** class, once an object of type **Color** is constructed, the object provides numerous methods and attributes that can be used to manipulate the color values. You will see later how the **r** and **g** attribute values are manipulated at runtime to cause the color of the rectangle's border to morph from red to green.

One of the overloaded constructors accepts integer parameters for the **red** , **green** , **blue** , and **alpha** color components (*also known as channels*) . As mentioned above, the **alpha** channel value defaults to 255 when not given.

The second and third statements in [Listing 3](#) create two different **Color** objects. One object, which is initialized to represent pure red (*the green and blue color values are set to zero*) will be used to specify the color of the rectangle's border. The red and green color values in this object will be modified at runtime to cause the color to morph from red to green.

The other **Color** object is initialized to represent pure green (*the red and blue color values are set to zero*) and will be used to specify the background color used to fill the display surface.

Create and set display mode on Surface object

According to the [documentation](#), *"A pygame Surface is used to represent any image... Call pygame.Surface() to create a new image object. The Surface will be cleared to all black. The only required arguments are the sizes. With no additional arguments, the Surface will be created in a format that best matches the display Surface."*

There are at least two different ways to create a **Surface** object. One way is to call the constructor for the **Surface** class as described above. The second way is to call the **pygame.display.set_mode** function, which is the way that will be used in this program. Once the **Surface** object is created, numerous methods and attributes are available to manipulate the surface.

According to the [documentation](#), *"This (pygame.display.set_mode) function will create a display Surface. The arguments passed in are requests for a display type. The actual created display will be the best possible match supported by the system."* The formal argument list contains three arguments with default values.

Continuing with the [documentation](#), *"The Surface that gets returned can be drawn to like a regular Surface but changes will eventually be seen on the monitor."* .

The fourth statement in [Listing 3](#) calls the `pygame.display.set_mode` function to create a new **Surface** object, connect it to the display, and set the width and height to 250 x 150 pixels respectively. A rectangle will be drawn on this Surface object and *"flipped"* to the screen during each iteration of the runtime loop.

I discussed other possible parameters to the `pygame.display.set_mode` function, (such as `FULLSCREEN`, `RESIZABLE`, and `NOFRAME`) in an earlier module. None of those parameters were used in this program.

An object of type Clock

According to the [documentation](#), a call to the `pygame.time.Clock` constructor *"Creates a new Clock object that can be used to track an amount of time. The clock also provides several functions to help control a game's framerate."*

The last statement in [Listing 3](#) creates a **Clock** object. The `tick` method of the **Clock** object will be called at the end of each iteration of the runtime loop to limit the program to no more than **FRAMERATE** frames per second. (*FRAMERATE is a constant that was defined in [Listing 2](#).*)

Enter the runtime loop

The statement in [Listing 4](#) causes the program to enter the runtime loop. Control will remain in the runtime loop until the value of the variable named `quit` changes from **False** to **True**. You have seen code like this in earlier modules.

Listing 4. Enter the runtime loop.

```
while not quit:
```

Get and process user inputs

You have seen code like that shown in [Listing 5](#) before without much in the way of an explanation. I will attempt to explain that code more fully in this module.

Listing 5. Get and process user inputs.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        quit = True #Make this the final iteration of runtime loop.
```

According to the [documentation](#),

"Pygame handles all its event messaging through an event queue. The routines in this module help you manage that event queue..."

The queue is a regular queue of [pygame.event.EventType](#) event objects, there are a variety of ways to access the events it contains. From simply checking for the existence of events, to grabbing them directly off the stack.

An [EventType](#) event object contains an event type identifier and a set of member data."

The code in [Listing 5](#) calls `pygame.event.get()`, which removes all of the [EventType](#) objects from the event queue and returns them in a list. The code uses a **for** loop to iterate through the list and to examine each object to determine if it is type QUIT.

Each event type is identified by a numeric value. Once you have access to an object of type [EventType](#), you can determine its type by calling the **type** method on the object, which returns the type as a numeric integer.

To eliminate the requirement for us to remember the numeric values for each event type, the **pygame** module defines a set of numeric constants with appropriate corresponding names and values. For example, the constant **pygame.QUIT** has a value that is equal to the numeric value for the event that occurs when the user clicks the X-button in the upper-right corner of the image in [Figure 1](#). This makes it possible for us to test for equality against a named constant to determine if an event is of a particular type.

The code in [Listing 5](#) tests each event from the queue to see if it matches **pygame.QUIT** and if so, changes the value of the variable named **quit** from **False** to **True**. This assures that the runtime loop will exit when the test is made at the beginning of the next iteration.

The names of constants corresponding to other types of events are listed below along with the attributes defined for each type of event.

- QUIT none
- ACTIVEEVENT gain, state
- KEYDOWN unicode, key, mod
- KEYUP key, mod
- MOUSEMOTION pos, rel, buttons
- MOUSEBUTTONUP pos, button
- MOUSEBUTTONDOWN pos, button
- JOYAXISMOTION joy, axis, value
- JOYBALLMOTION joy, ball, rel
- JOYHATMOTION joy, hat, value
- JOYBUTTONUP joy, button
- JOYBUTTONDOWN joy, button
- VIDEORESIZE size, w, h
- VIDEOEXPOSE none
- USEREVENT code

You can read more about these event types in the [documentation](#).

This module will deal only with the simple QUIT event type. Future modules will deal with more complex event types such as KEYDOWN and MOUSEBUTTONDOWN.

The code in [Listing 6](#) calls the **fill** method of the **Surface** object to cause it to take on the color represented by the **Color** object referred to by the variable named **background** . Recall from [Listing 3](#) that this **Color** object represents pure green.

Listing 6. Fill display Surface object with green background Color object.

```
surface.fill(background)
```

The **fill** method has other default parameters that are not used here. I may explore some of them in future modules.

Cause border Color object to morph from red to green

The code in [Listing 7](#) causes the color represented by the **Color** object referred to by the variable named **border** to morph from red to green on successive iterations of the runtime loop. When it reaches green, it resets back to red.

Listing 7. Cause border Color object to morph from red to green.

```
if border.r >= INCREMENT:
    border.r -= INCREMENT #Decrease red color component
    border.g += INCREMENT #Increase green color component
else: #Reset the border Color object to pure red
    border = pygame.Color(255,0,0)
```

The **Color** object referred to by **border** has attributes named **r** , **g** , and **b** . These attributes represent the red, green, and blue color channels respectively. They can be accessed and modified at runtime to cause the state of the object to change at runtime.

The morphing effect is accomplished by decreasing the value of the attribute named **r** and increasing the value of the attribute named **g** by the value stored in **INCREMENT** once during each iteration of the runtime loop.

Values for **r** and **g** less than zero or greater than 255 are not allowed so logic is provided to prevent that from happening. When the value of **r** becomes less than the value of **INCREMENT**, the **else** clause causes the modified **Color** object to be replaced by a new **Color** object that represents pure red and the cycle repeats.

*This reset could also have been accomplished by setting **r** to 255 and setting **g** to 0 without creating a new object.*

Draw and display the rectangle

The code in [Listing 8](#) draws a rectangle on the display **Surface** object passing a **Color** object and a **Rect** object as parameters. The constant named **WIDTH** is also passed as a parameter causing the border width to be 25 pixels.

Listing 8. Draw and display the rectangle.

```
pygame.draw.rect(surface, border, rect, WIDTH)

pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame rate in frames per second
```

This code differs from code in an earlier module in that **Color** and **Rect** objects are passed as parameters to **pygame.draw.rect** instead of passing tuples. This illustrates the overloaded nature of **pygame.draw.rect**. In my opinion, it also makes the program more self-documenting.

[Listing 8](#) also calls **flip**, which you learned about in an earlier module, to cause the contents of the display surface to be copied to the screen.

Finally, [Listing 8](#) calls the **tick** method, which you also learned about in an earlier module to insert a time delay to limit the frame rate of the program to less than FRAMERATE in frames per second.

At this point, control returns to the top of the runtime loop, which will either exit or begin a new iteration depending of the value stored in the variable named **quit**.

Terminate the program outside the runtime loop

The code in [Listing 9](#) terminates the program after control exits the runtime loop.

Listing 9. Terminate the program outside the runtime loop.

```
pygame.quit() #Terminate the program outside the runtime loop
```

There is nothing new with this code so an explanation should not be needed.

Run the program

I encourage you to copy the code from [Listing 10](#). Execute the code and confirm that you get the same results as those shown in [Figure 1](#) through [Figure 3](#).. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is provided in [Listing 10](#) below.

Listing 10. Complete program listing.

Listing 10. Complete program listing.

```
"""
File Color01.py
Revised: 01/03/15
Draws a rectangle with a wide colored border on a green background. The border
color starts as red and slowly morphs into green. When it reaches green, it
switches back to red and the cycle continues.
=====
"""

import pygame #Import required library

pygame.init() #Initialize imported pygame modules
quit = False #Initialize termination control variable.
pygame.display.set_caption("Color01.py") #Set the window caption
INCREMENT = 20 #Initialize control constant
WIDTH = 25 #Initialize border width constant
FRAMERATE = 5 #Used to control the maximum frame rate.

rect = pygame.Rect(25, 25, 200, 100) #create a Rect object
border = pygame.Color(255,0,0) #Create a red Color object
background = pygame.Color( 0, 255, 0) #Create a green Color object
surface = pygame.display.set_mode([250,150]) #Set the display mode
clock = pygame.time.Clock() #Create an object to help track time

#Enter the runtime loop
while not quit:
    #Get user inputs for this iteration of the runtime loop.
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final iteration of runtime loop.

    #Fill display Surface object with green background Color object.
    surface.fill(background)

    #Cause border Color object to morph from red to green
    if border.r >= INCREMENT:
        border.r -= INCREMENT #Decrease red color component
        border.g += INCREMENT #Increase green color component
    else: #Reset the border Color object to pure red
        border = pygame.Color(255,0,0)

    #Draw a rectangle on the display Surface object with border Color object
    # and Rect object. Make the border width 25 pixels.
    pygame.draw.rect(surface, border, rect, WIDTH)

    #Display the new state
    pygame.display.flip()

    clock.tick(FRAMERATE) #Control the frame rate in frames per second

pygame.quit() #Terminate the program outside the runtime loop
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2220-Color-Part 1
- File: Itse1359-2220.htm
- Published: 01/04/15
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2225-Color-Introduction to Alpha Transparency

This module introduces the student to the use of alpha transparency in Pygame.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Create some constants and variables](#)
 - [Create some objects](#)
 - [Enter the runtime loop](#)
 - [Create some colored rectangular areas on the base surface](#)
 - [Create a new surface](#)
 - [Draw the new surface on the base surface](#)
 - [Display the surface and control the clock](#)
 - [Terminate the program](#)
- [Run the program](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Program output.

Listings

- [Listing 1.](#) The preliminaries.
- [Listing 2.](#) Create some constants and variables.
- [Listing 3.](#) Create some objects.
- [Listing 4.](#) Enter the runtime loop.
- [Listing 5.](#) Create some colored rectangular areas on the base surface.
- [Listing 6.](#) Create a new surface.
- [Listing 7.](#) Draw the new surface on the base surface.
- [Listing 8.](#) Terminate the program.
- [Listing 9.](#) Complete program listing.

General background information

One of the more complicated aspects of dealing with color has to do with *transparency* or its inverse, *opacity* . There are at least [three types of transparency](#) supported in Pygame:

- colorkeys
- surface alphas
- pixel alphas

This module will deal with *surface alphas* and defer a discussion of the other two types to future a module.

A surface alpha value is a single value that changes the transparency for the entire image. A surface alpha value of 255 is opaque, and a value of 0 is completely transparent.

Although there may be some exceptions, alpha values belonging to pixels written onto the base surface created by calling `pygame.display.set_mode` will be ignored. For example, if you call `python.draw.rect` to draw a rectangle on the base surface and set the alpha value for the border color to 128 (50%) , the border will still be opaque.

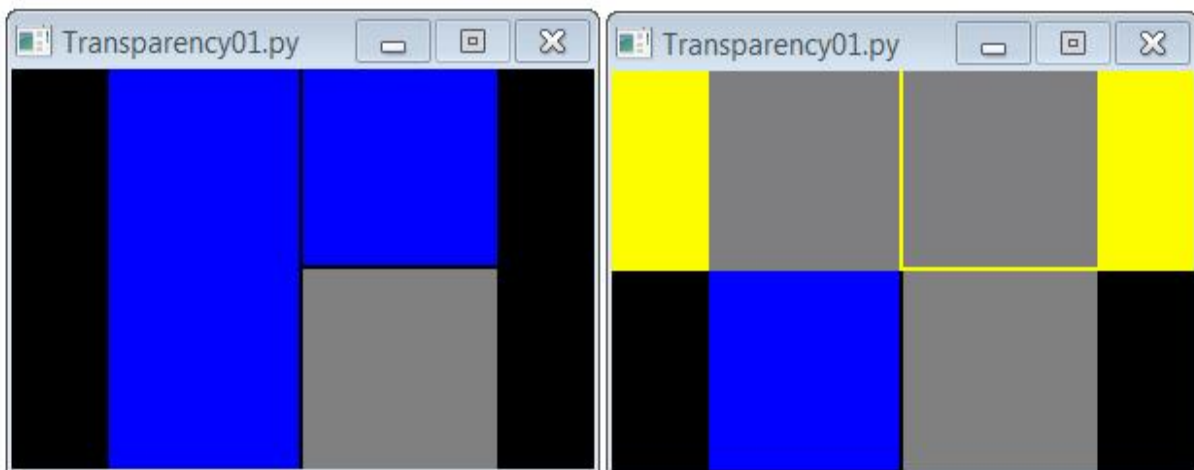
To make use of the alpha channel, you will need to

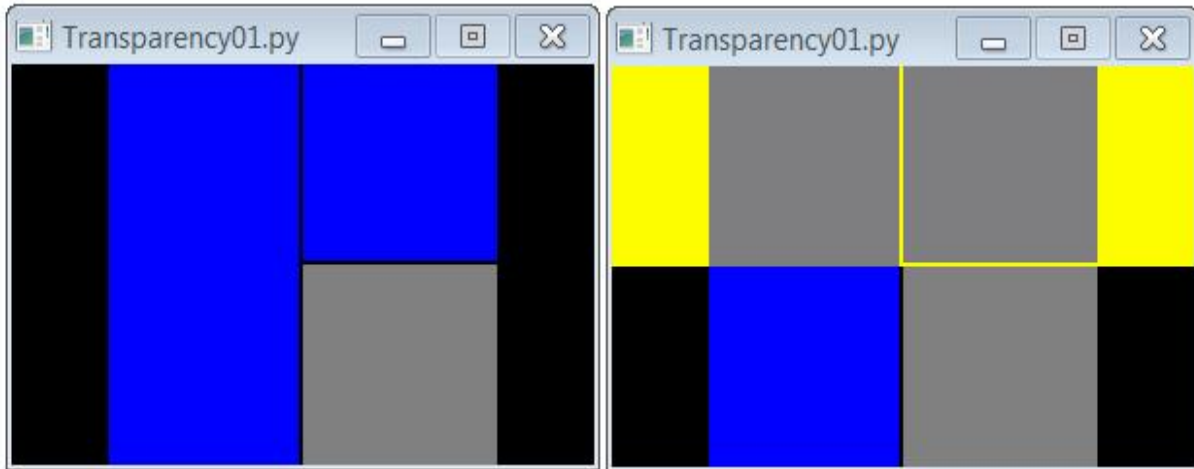
- create a surface that supports the alpha channel,
- manipulate the color data on that surface, and
- [blit](#) that surface to the base surface.

There are several ways to do that. I will show you one of those ways in this module and will show you other ways in future modules.

The program that I will explain in the [next section](#) begins by creating two blue and one gray rectangular areas on the base surface as shown by the image in the left half of [Figure 1](#). (Note that the default surface background is black.)

Figure 1. Program output.





Then the program creates a new surface with a format that contains surface alphas. The surface alpha value is set to 127.5 (*the decimal part is probably ignored for an alpha value of 127 or 49.8-percent opaque*) . The width of the new surface is equal to the width of the base surface and the height is half the height of the base surface. The new surface is drawn onto the upper half of the base surface by calling the [blit](#) method of the [Surface](#) class.

This produces the output shown in the right half of [Figure 1](#) . Blending the yellow color with the blue color using the default blending formula produces the gray color shown in the top half of the right half of [Figure 1](#) . To the eye, at least, this gray color matches the gray color of the bottom-right rectangle in both images.

There are many different formulas that can be used to compute the resulting color when two colors are blended in pygame. They have names like BLEND_ADD and BLEND_SUB and can be passed as **special_flags** to the **blit** method.

This program uses the default blending formula. The output shown in [Figure 1](#) can be used to make an educated guess that the behavior of the default blending formula is as follows.

If the color values of the surface being copied to are represented by vd (*for destination*) , the color values of the surface being copied from are represented by vs (*for source*) and the resulting color values are represented by vr , then it appears that

$$vr = vs * (\alpha/255) + vd * (1 - (\alpha/255))$$

(I will show in a future module that this is apparently not true if all three color values on the surface being copied to are zero indicating that the source is being copied to a black surface. In that case, the resulting color values appear to match the source color values regardless of the value of alpha.)

Discussion and sample code

A complete listing of this program is provided in [Listing 9](#). I will break the program down and explain it in fragments. The first fragment is shown in [Listing 1](#).

Listing 1. The preliminaries.

```
import pygame #Import required library
pygame.init() #Initialize imported pygame
modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Transparency01.py")
#Set the window caption
```

I have explained code like that shown in [Listing 1](#) in earlier modules. Therefore, no further explanation of this code should be needed.

Create some constants and variables

In keeping with my philosophy that we should strive to make our code as self-documenting as practical, this program makes use of variables with meaningful names as opposed to simply sticking numbers in various locations. Five such variables are created in [Listing 2](#). You will learn what these variables are used for later in the module.

Listing 2. Create some constants and variables.

```
FRAMERATE = 4 #Used to control the maximum  
frame rate.  
WIDTH = 300  
HEIGHT = 200  
ORIGIN = (0,0)  
fiftyPercent = 255/2
```

Create some objects

Once again, in keeping with my philosophy regarding self documenting code, this program uses objects referred to by variables with meaningful names. Several such objects are created in [Listing 3](#). You will see what these objects are used for later. Note that the second statement in [Listing 3](#) uses the variables WIDTH and HEIGHT that were created in [Listing 2](#).

Listing 3. Create some objects.

```
#Create and set the display mode on the base
surface.
baseSurf =
pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object
to help track time

#Create some color objects
blue = pygame.Color(0,0,255)
gray = pygame.Color(128,128,128)
yellow = pygame.Color(255,255,0)

#Create some Rect objects for use in
specifying rectangular areas.
leftRectArea = pygame.Rect(50,0,98,200)
upperRightRectArea = pygame.Rect(150,0,100,98)
lowerRightRectArea =
pygame.Rect(150,100,100,100)
```

Enter the runtime loop

The code in [Listing 4](#) enters the runtime loop and checks for user input.

Listing 4. Enter the runtime loop.

Listing 4. Enter the runtime loop.

```
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.
```

I have explained code like this in earlier modules and a further explanation should not be needed.

Create some colored rectangular areas on the base surface

There are at least two ways to create filled rectangles in pygame. One way is to call [pygame.draw.rect](#) and set the border width to zero. The other way is to

- call the [fill](#) method on a [Surface](#) object,
- specify the color as the first parameter, and
- specify a rectangular area as the second parameter.

That is the approach that was used in this program.

Listing 5. Create some colored rectangular areas on the base surface.

Listing 5. Create some colored rectangular areas on the base surface.

```
baseSurf.fill(blue, leftRectArea) #Blue
area on left
baseSurf.fill(blue, upperRightRectArea)
#Blue square - upper right
baseSurf.fill(gray, lowerRightRectArea)
#Gray square - lower right
```

Taken together, the three statements in [Listing 5](#) produce the three colored rectangular areas shown in the left half of [Figure 1](#). Note that these statements pass **Color** and **Rect** object references that were created in [Listing 3](#) as parameters to the **fill** method..

Create a new surface

[Listing 6](#) creates a new surface with 50-percent transparency (*also 50-percent opacity*) and a yellow color half the size of the base surface.

Listing 6. Create a new surface.

```
s1 = pygame.Surface((WIDTH, HEIGHT/2))
s1.set_alpha(fiftyPercent)
s1.fill(yellow)
```

[Listing 6](#) begins by calling one of the overloaded constructors for the [Surface](#) class passing width and height information in a tuple as the first parameter. As near as I can tell, the documentation does not explicitly state that the resulting surface format contains surface alphas. However, the documentation for the [set_alpha](#) method states *"If the Surface format contains per pixel alphas, then this alpha value will be ignored."* The documentation also states *"Per pixel alphas cannot be mixed with surface alpha and colorkeys."*

The fact that the alpha value that is set in [Listing 6](#) is not ignored strongly suggests that the default surface format does not contain per pixel alphas and does contain surface alphas.

[Listing 6](#) calls the **fill** method on the **Surface** object to set the color of the entire surface to **yellow** , as defined in [Listing 3](#).

Draw the new surface on the base surface

Calling the [blit](#) method on a surface causes the surface passed as the first parameter (*source*) to be drawn on the surface on which the method is called. The second parameter specifies the location at which the upper-left corner of the source will be drawn. In this case, the upper-left corners of the two surfaces were aligned. There are other parameters that can be passed to control other aspects of the drawing as well, but they weren't used in this case.

Running the code in [Listing 7](#) with the # character intact disables the call to the **blit** method and produces the output shown in the left half of [Figure 1](#). Removing the # character and running the code in [Listing 7](#) produces the output shown in the right half of [Figure 1](#).

Listing 7. Draw the new surface on the base surface.

```
#Draw the new surface at the origin on the
baseSurf.
#Disable the following statement to see
the raw baseSurf.
#    baseSurf.blit(s1,ORIGIN)

#Display the baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame
rate in frames per second
```

Display the surface and control the clock

The remaining code in [Listing 7](#) is the same as code that I have explained in earlier modules. Therefore, no further explanation of this code should be needed.

Terminate the program

The code in [Listing 8](#) does any necessary cleanup and terminates the program when the user clicks the X-button in the upper-right of [Figure 1](#), causing control to exit the runtime loop.

Listing 8. Terminate the program.

```
pygame.quit() #Terminate the program outside  
the runtime loop
```

Run the program

I encourage you to copy the code from [Listing 9](#). Execute the code and confirm that you get the same results as those shown in [Figure 1](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listings

A complete listing of the program is shown in [Listing 9](#).

Listing 9. Complete program listing.

```
"""  
File Transparency01.py  
Illustrates surface alpha blending.  
=====
```

```
=====
```

```
"""  
import pygame #Import required library  
pygame.init() #Initialize imported pygame modules  
  
quit = False #Initialize termination control  
variable.  
pygame.display.set_caption("Transparency01.py")  
#Set the window caption  
FRAMERATE = 4 #Used to control the maximum frame  
rate.
```

```

WIDTH = 300
HEIGHT = 200
ORIGIN = (0,0)
fiftyPercent = 255/2

#Create and set the display mode on the base
surface.
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object to
help track time

#Create some color objects
blue = pygame.Color(0,0,255)
gray = pygame.Color(128,128,128)
yellow = pygame.Color(255,255,0)

#Create some Rect objects for use in specifying
rectangular areas.
leftRectArea = pygame.Rect(50,0,98,200)
upperRightRectArea = pygame.Rect(150,0,100,98)
lowerRightRectArea = pygame.Rect(150,100,100,100)

while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.

        #Create some colored rectangular areas on the
baseSurf
        baseSurf.fill(blue,leftRectArea) #Blue area on
left
        baseSurf.fill(blue,upperRightRectArea) #Blue
square - upper right
        baseSurf.fill(gray,lowerRightRectArea) #Gray
square - lower right

```

```
#Create a new surface with 50%
transparency/opacity and a yellow color
# half the size of baseSurf.
s1 = pygame.Surface((WIDTH,HEIGHT/2))
s1.set_alpha(fiftyPercent)
s1.fill(yellow)

#Draw the new surface at the origin on the
baseSurf.
#Disable the following statement to see the
raw baseSurf.
#    baseSurf.blit(s1,ORIGIN)

#Display the baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame rate
in frames per second

pygame.quit() #Terminate the program outside the
runtime loop
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2225-Color-Part 2
- File: Itse1359-2225.htm
- Published: 01/31/16
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2230-Color-Animated Demonstration of Surface Alphas
This module presents an animated demonstration of surface alphas.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [The program output](#)
 - [Dynamic behavior of the program](#)
- [Discussion and sample code](#)
 - [The preliminaries](#)
 - [Enter the runtime loop](#)
 - [The program logic](#)
 - [The remainder of the runtime loop code](#)
 - [Terminate the program](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Screen shots from a dynamic demonstration of runtime alphas.

Listings

- [Listing 1.](#) The preliminaries.
- [Listing 2.](#) Enter the runtime loop.
- [Listing 3.](#) The program logic.
- [Listing 4.](#) The remainder of the runtime loop code.
- [Listing 5.](#) Terminate the program.
- [Listing 6.](#) Complete program listing.

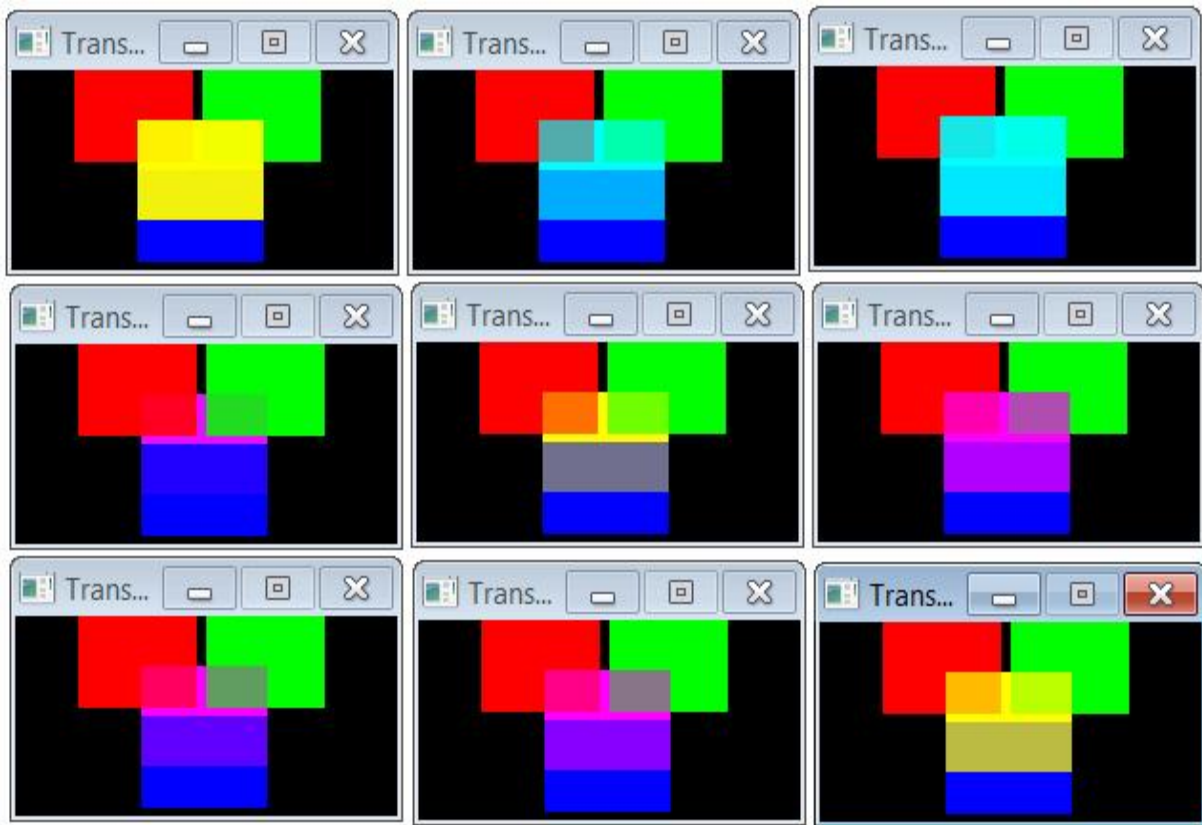
General background information

In an earlier module, you learned about the runtime loop and how code that is executed within the runtime loop can cause the display to change over time. You also learned about *surface alphas* in an earlier module. In this module, you will learn how to write a program that displays a dynamic demonstration of surface alphas that change over time.

The program output

[Figure 1](#) shows nine screen shots taken at random times while the program was running.

Figure 1. Screen shots from a dynamic demonstration of runtime alphas.



The program creates three colored rectangular areas (*red, green, and blue*) on the black base surface during each iteration of the runtime loop. The red and green areas are shown side by side at the top of each output image in [Figure 1](#). The blue area is shown centered below the red and green areas.

Also, during each iteration, the program creates a small, partially transparent rectangular surface with a color of yellow, cyan, or magenta and draws it at the center of the display on top of the red, green, and blue areas. (*For want of a better term, I will refer to this area as the mask.*)

You can see the actual color of the mask where it overlaps the black areas that separate the red, green, and blue areas. (*The mask displays its actual color, or something very close to its actual color in those black areas even for alpha values as low as 16.*)

The top left image in [Figure 1](#) shows the mask during a period when it is yellow with a high alpha value. The top center image shows the mask when it is cyan with a moderate alpha value. The top right image shows the mask when it is cyan with a moderate relatively high value.

The images at each end of the center row shows the mask when it is magenta with a low alpha value on the left and a somewhat higher value on the right. None of the images show magenta with a really high alpha value.

The important thing to note in [Figure 1](#) is the blended colors produced by the overlap of the mask with the red, green, and blue areas. However, the results are much more instructive when viewed in real time while the program is running.

Dynamic behavior of the program

When the program starts, the mask has a yellow color with an alpha value of zero (*totally transparent*) . During each iteration of the runtime loop, the alpha value is increased until it reaches 255 (*totally opaque*) . When the alpha value of the yellow mask reaches 255, the color of the mask is switched to cyan and the alpha value is set to zero.

As the iterations continue, the alpha value increases again until it reaches 255. When that happens, the color of the mask is switched from cyan to magenta and the alpha value is once again set to zero.

The process continues as before. When the alpha value of the magenta mask reaches 255, the color is switched back to yellow with an alpha value of zero. This cycle repeats until the user clicks the X-button in the upper-left corner of the output image shown in [Figure 1](#).

Discussion and sample code

The preliminaries

A complete listing of the program is provided in [Listing 6](#) near the end of the module. I will break the program down and explain it in fragments. The first fragment is shown in [Listing 1](#).

Listing 1. The preliminaries.

```
import pygame #Import required library
pygame.init() #Initialize imported pygame
modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Transparency02.py")
#Set the window caption
FRAMERATE = 40 #Used to control the maximum
frame rate.
WIDTH = 200
HEIGHT = 100
ORIGIN = (WIDTH/3,HEIGHT/4)
alpha = 0
alphaInc = 1
colorNumber = 0

#Create and set the display mode on the base
surface.
baseSurf =
pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object
to help track time

#Create some color objects
```

Listing 1. The preliminaries.

```
red = pygame.Color(255,0,0)
green = pygame.Color(0,255,0)
blue = pygame.Color(0,0,255)
yellow = pygame.Color(255,255,0)
cyan = pygame.Color(0,255,255)
magenta = pygame.Color(255,0,255)

#Populate a tuple with references to the color
objects
colors = (yellow,cyan,magenta)

#Create some Rect objects for use in specifying
rectangular areas.
leftRectArea = pygame.Rect(WIDTH/6,0,WIDTH/3-
4,HEIGHT/2-4)
rightRectArea = pygame.Rect(WIDTH/2,0,WIDTH/3-
4,HEIGHT/2-4)
bottomRectArea =
pygame.Rect(WIDTH/3,HEIGHT/2,WIDTH/3,HEIGHT/2-
4)
```

Almost all of the code in [Listing 1](#) is similar to code that you have seen in previous modules so a detailed explanation should not be needed. However, there are a couple of things that are worthy of note.

First, all coordinate, width, and height values in the program are specified in terms of the values contained in the constants named WIDTH and HEIGHT. Therefore, you can change the size of the display and its contents simply by changing those values. In order to make it possible to show nine screen shots in a small amount of space, those values were set relatively small to create the screen shots in [Figure 1](#). You may want to increase those values when you copy the code from [Listing 6](#) and run the program to provide a better view of the results.

Second, after creating **Color** objects for **yellow** , **cyan** , and **magenta** , references to those objects are stored in a tuple referred to by the variable named **colors** . This tuple will be indexed inside the runtime loop to specify the color of the mask for the current iteration.

Enter the runtime loop

The code in [Listing 2](#) is also very similar to code that I have explained in earlier modules. Therefore, a detailed explanation should not be needed.

Listing 2. Enter the runtime loop.

```
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.

    #Create some colored rectangular areas on
baseSurf
    baseSurf.fill(red,leftRectArea) #Red area
on left
    baseSurf.fill(green,rightRectArea) #Green
area on the right
    baseSurf.fill(blue,bottomRectArea) #Blue
area on the bottom
```


The program logic

[Listing 3](#) uses relatively straightforward logic to compute the alpha value (*alpha*) and the color index (*colorNumber*) that will be applied to the mask for this iteration of the runtime loop.

Listing 3. The program logic.

```
if alpha <= (255 - alphaInc):  
    alpha += alphaInc  
else:  
    alpha = 0  
    colorNumber += 1  
    if colorNumber > 2:  
        colorNumber = 0
```

If you have learned about Python *flow of control* , you should have no difficulty understanding the code in [Listing 3](#) without a detailed explanation.

The remainder of the runtime loop code

The code in [Listing 4](#) creates a new surface for the mask using the values of the **WIDTH** and **HEIGHT** constants to specify the dimensions of the surface.

Listing 4. The remainder of the runtime loop code.

```
s1 =  
pygame.Surface((2*WIDTH/6, 2*HEIGHT/4))  
s1.set_alpha(alpha)  
s1.fill(colors[colorNumber])  
  
#Draw the new surface on baseSurf at the  
specified location.  
baseSurf.blit(s1, ORIGIN)  
  
#Display baseSurf  
pygame.display.flip()  
  
clock.tick(FRAMERATE) #Control the frame  
rate in frames per second
```

Then the code in [Listing 4](#) sets the alpha value for the surface using the value contained in the variable named **alpha** that was computed in [Listing 3](#) .

After that, the code uses the value of the variable named **colorNumber** , which was computed in [Listing 3](#) , to index the tuple named **colors** to get the color used to **fill** the mask.

The code in [Listing 4](#) calls the **blit** method to draw the mask on the base surface at a location specified by the value of the constant named **ORIGIN** .

Finally, the code calls the **tick** method on the **Clock** object to insert a delay and assure that the frame rate will be less than or equal to the value stored in the constant named **FRAMERATE** .

Terminate the program

When the user clicks the X-button shown in [Figure 1](#), control will exit the runtime loop. This causes the code shown in [Listing 5](#) to be executed, which will perform any necessary cleanup and terminate the program.

Listing 5. Terminate the program.

```
pygame.quit() #Terminate the program outside  
the runtime loop
```

Run the program

I encourage you to copy the code from [Listing 6](#). Execute the code and confirm that you get results similar to those shown in [Figure 1](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is shown in [Listing 6](#).

Listing 6. Complete program listing.

```
"""  
File Transparency02.py  
Presents a dynamic demonstration of surface  
alphas.  
=====
```

```

=====
"""
import pygame #Import required library
pygame.init() #Initialize imported pygame modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Transparency02.py")
#Set the window caption
FRAMERATE = 40 #Used to control the maximum frame
rate.
WIDTH = 200
HEIGHT = 100
ORIGIN = (WIDTH/3,HEIGHT/4)
alpha = 0
alphaInc = 1
colorNumber = 0

#Create and set the display mode on the base
surface.
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object to
help track time

#Create some color objects
red = pygame.Color(255,0,0)
green = pygame.Color(0,255,0)
blue = pygame.Color(0,0,255)
yellow = pygame.Color(255,255,0)
cyan = pygame.Color(0,255,255)
magenta = pygame.Color(255,0,255)

#Populate a tuple with references to the color
objects
colors = (yellow,cyan,magenta)

#Create some Rect objects for use in specifying

```

```

rectangular areas.
leftRectArea = pygame.Rect(WIDTH/6,0,WIDTH/3-
4,HEIGHT/2-4)
rightRectArea = pygame.Rect(WIDTH/2,0,WIDTH/3-
4,HEIGHT/2-4)
bottomRectArea =
pygame.Rect(WIDTH/3,HEIGHT/2,WIDTH/3,HEIGHT/2-4)

#Enter the runtime loop.
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.

        #Create some colored rectangular areas on
baseSurf
        baseSurf.fill(red,leftRectArea) #Red area on
left
        baseSurf.fill(green,rightRectArea) #Green area
on the right
        baseSurf.fill(blue,bottomRectArea) #Blue area
on the bottom

        #Compute alpha value and color for this
iteration of the runtime loop.
        if alpha <= (255 - alphaInc):
            alpha += alphaInc
        else:
            alpha = 0
            colorNumber += 1
            if colorNumber > 2:
                colorNumber = 0

        #Create a new surface smaller than baseSurf.
Set the alpha and the color.
        s1 = pygame.Surface((2*WIDTH/6,2*HEIGHT/4))

```

```
s1.set_alpha(alpha)
s1.fill(colors[colorNumber])

#Draw the new surface on baseSurf at the
specified location.
baseSurf.blit(s1,ORIGIN)

#Display baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame rate
in frames per second

pygame.quit() #Terminate the program outside the
runtime loop
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2230-Color-Part 3
- File: Itse1359-2230.htm
- Published: 01/31/16
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2235-Color-Animated Demonstration of Per Pixel Alpha Blending

This module provides an animated demonstration of per pixel alpha blending with a draw function.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [The program output](#)
 - [Behavior of the program](#)
- [Discussion and sample code](#)
 - [The preliminaries](#)
 - [Enter the runtime loop](#)
 - [Create a Color object for the circle](#)
 - [Create a Color object for the lower right quadrant](#)
 - [Draw three filled rectangles on the base surface](#)
 - [Compute the alpha value for the circle](#)
 - [Create a new surface with a format that contains per pixel alphas](#)
 - [Draw a circle on the mask](#)
 - [The remainder of the code](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Screen shots from an animated demonstration of per pixel alphas that change over time.

Listings

- [Listing 1.](#) The preliminaries.
- [Listing 2.](#) Enter the runtime loop.
- [Listing 3.](#) Create a Color object for the circle.
- [Listing 4.](#) Create a Color object for the lower right quadrant.
- [Listing 5.](#) Draw three filled rectangles on the base surface.
- [Listing 6.](#) Compute the alpha value for the circle.
- [Listing 7.](#) Create a new surface with a format that contains per pixel alphas.
- [Listing 8.](#) Draw a circle on the mask.
- [Listing 9.](#) The remainder of the code.
- [Listing 10.](#) Complete program listing.

General background information

In an earlier module, you learned about the runtime loop and how code that is executed within the runtime loop can cause the display to change over time. You also learned about *surface alphas* in an earlier module. With surface alphas, a single alpha value applies to every pixel on the surface. Changing the surface alpha value changes the transparency/opacity of the entire surface.

In this module, you will learn about *per pixel alphas*. In other words, you will learn how to write code where the alpha value of each pixel may be different from the alpha values of its neighbors. Those alpha values can be changed at runtime independently of one another.

You will learn how to compute the red, green, and blue color values produced by the default blending formula when a color with a given alpha value is drawn over another color.

You will learn how to write a program that displays an animated demonstration of per pixel alphas that change over time.

Finally, you will learn how to use two of the functions from the [pygame.draw](#) module for drawing a [circle](#) and for drawing several [rectangles](#). (Note that this is different from using the *fill* method to create colored rectangular areas on the base surface as was done in an

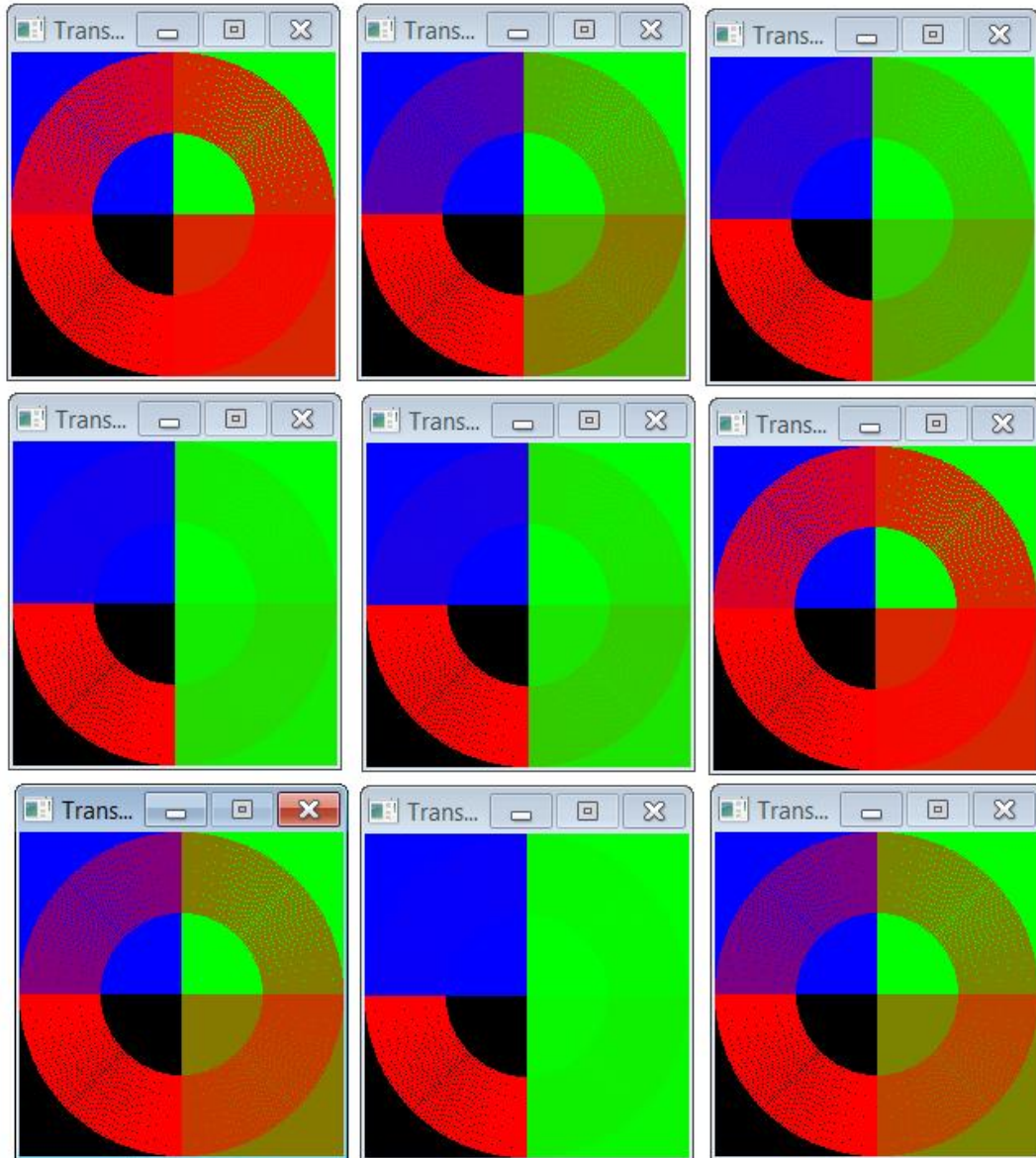
earlier module.) Other functions in the [pygame.draw](#) module can be used to draw polygons, ellipses, arcs, and various forms of lines.

(Be careful to avoid confusing Python/Pygame *modules* with cnx.org *modules* . A Python/Pygame module is a programming construct of the Python programming language. A cnx.org module is a document in the cnx legacy view or a page in the openstax view.)

The program output

[Figure 1](#) shows nine screen shots taken at random times while the program was running.

Figure 1. Screen shots from an animated demonstration of per pixel alphas that change over time.



Behavior of the program

During each iteration of the runtime loop, the program draws three filled rectangles on the base surface and draws a circle on a surface that supports per pixel alphas. For want of a better term, I will refer to this second surface as the *mask*. After drawing the circle on the mask, the program blits the mask onto the base surface and then displays the base surface.

A blue rectangle is drawn in the upper left quadrant of the base surface as shown by all of the screen shots in [Figure 1](#). A green rectangle is drawn in the upper right quadrant as also shown by all of the screen shots in [Figure 1](#). The colors of these two rectangles do not change over time.

Nothing is drawn in the lower left quadrant of the base surface leaving it constantly black.

A red circle with a wide border and an alpha value that changes over time is drawn on the mask. The circle is centered on the mask as shown by the screen shots in Figure 1.

A rectangle is also drawn in the lower right quadrant of the base surface. The color of this rectangle changes over time and is designed to match the color produced by blending the red circle with the green color in the upper right quadrant. For example, the color of the lower right quadrant in the rightmost screen shot on the third row of [Figure 1](#) is a color that we might refer to as tan. This is the same color produced by the overlap of the red circle with the green color in the upper right quadrant.

The alpha value for the red circle starts at zero (*totally transparent*) and increases during each iteration of the runtime loop until it reaches 255 (*totally opaque*) . Then it resets back to zero. During this process, the blended overlap region in the upper right quadrant takes on 256 different colors ranging from green to red with many different colors in between. The color of the lower right rectangle tracks the color of the overlap region in the upper right rectangle.

The red circle appears to be red in the lower left black area for all but the smallest values of alpha.

The overlap region in the blue upper left rectangle progresses from blue for an alpha value of zero to red for an alpha value of 255. Along the way, various violets, purples, magentas, etc., are produced.

The overlap region in the bottom right rectangle progresses from green for an alpha value of zero to red for an alpha value of 255. Many different colors are produced in between those extremes.

Discussion and sample code

The preliminaries

A complete listing of the program is provided in [Listing 10](#) near the end of the module. I will break the program down and explain it in fragments. The first fragment is shown in [Listing 1](#).

Listing 1. The preliminaries.

Listing 1. The preliminaries.

```
import pygame #Import required library
pygame.init() #Initialize imported pygame modules

quit = False #Initialize termination control variable.
pygame.display.set_caption("Transparency03.py") #Set the
window caption
FRAMERATE = 20 #Used to control the maximum frame rate.
WIDTH = 200 #Width of output window
HEIGHT = 200 #Height of output window
RADIUS = WIDTH//2 #Radius of circle
BORDERWIDTH = RADIUS//2 #Width of circle border
ORIGIN = (0,0) #Location of the mask on the base surface
alpha = 0 #Initial opacity of circle
alphaInc = 1 #Variable used to change opacity of circle

#Create and set the display mode on the base surface.
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object to help
track time

#Create some objects of type Rect to use later with
pygame.draw.rect
upperLeftRect = pygame.Rect(0,0,WIDTH/2,HEIGHT/2)
upperRightRect = pygame.Rect(WIDTH/2,0,WIDTH/2,HEIGHT/2)
lowerRightRect =
pygame.Rect(WIDTH/2,HEIGHT/2,WIDTH/2,HEIGHT/2)

#Create some color objects
upperLeftColor = pygame.Color(0,0,255) #Upper left color
upperRightColor = pygame.Color(0,255,0) #Upper right
corner
```

There is very little that is new in [Listing 1](#) so a detailed explanation should not be needed. It is worth noting however, that the values of some of the variables are computed using the integer form of division, such as:

```
RADIUS = WIDTH//2 #Radius of circle
```

This is required because the variable is later passed as a parameter to a method or function that requires the parameter to be of type integer.

All of the locations and dimensions in this program are specified in terms of the contents of the variables named `WIDTH` and `HEIGHT`. Therefore, you can change the overall size and dimensions of the output by changing the values of `WIDTH` and `HEIGHT`.

Enter the runtime loop

The code in [Listing 2](#) enters the runtime loop and tests for user input.

Listing 2. Enter the runtime loop.

```
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final iteration of
the runtime loop.
```

There is nothing new in [Listing 2](#).

Create a `Color` object for the circle

The statement in [Listing 3](#) creates a **`Color`** object for a given alpha value that will be used later to set the color for the circle.

Listing 3. Create a `Color` object for the circle.

```
circleColor = pygame.Color(255, 0, 0, alpha)
```

The statement in [Listing 3](#) must be inside the runtime loop because the value of alpha is different for each iteration of the runtime loop. As explained earlier, it starts at 0 and

increases during each iteration of the loop until it reaches 255. At that point it resets to 0 and the cycle repeats.

Create a **Color** object for the lower right quadrant

The code in [Listing 4](#) creates a **Color** object that matches the color that results from blending the circle color with the green color of the upper right quadrant. This **Color** object is used to set the color of the lower right quadrant in each screen shot in [Figure 1](#).

Listing 4. Create a **Color** object for the lower right quadrant.

```
lowerRightColor = pygame.Color(  
    int(circleColor.r*circleColor.a/255+  
        upperRightColor.r*(1-  
circleColor.a/255)),  
    int(circleColor.g*circleColor.a/255+  
        upperRightColor.g*(1-  
circleColor.a/255)),  
    int(circleColor.b*circleColor.a/255+  
        upperRightColor.b*(1-  
circleColor.a/255)))
```

The code in [Listing 4](#) probably needs some explanation. If the color values of the base surface are represented by redBase, greenBase, and blueBase and the color values of the circle are represented by redCircle, greenCircle, and blueCircle, the resulting blended color values can be computed as

```
redResult = redCircle*(alpha/255) + redBase*(1-alpha/255)  
greenResult = greenCircle*(alpha/255) + greenBase*(1-  
alpha/255)  
blueResult = blueCircle*(alpha/255) + blueBase*(1-alpha/255)
```

A **Color** object has several attributes including attributes for the red, green, blue, and alpha values. For a **Color** object referred to by a variable named **obj**, those attribute values can be accessed as follows:

```
red = obj.r  
green = obj.g
```

```
blue = obj.b  
alpha = obj.a
```

That is the notation used in [Listing 4](#) to access and perform arithmetic with the red, green, blue, and alpha values of the circle and the green filled rectangle drawn in the upper right quadrant on the base surface. The code in [Listing 4](#) uses that notation to access and compute the blended color values for the overlap between the circle and the green upper right quadrant. Those color values are then used to create a **Color** object that is used to set the color of the lower right quadrant. This causes the color of the lower right quadrant to track the overlap in the upper right quadrant.

Draw three filled rectangles on the base surface

The code in [Listing 5](#) calls `pygame.draw.rect` three times in succession to draw the three filled rectangles on the base surface passing **Color** objects and **Rect** objects created earlier as parameters..

Listing 5. Draw three filled rectangles on the base surface.

```
pygame.draw.rect(baseSurf, upperLeftColor, upperLeftRect, 0)  
pygame.draw.rect(baseSurf, upperRightColor, upperRightRect, 0)  
pygame.draw.rect(baseSurf, lowerRightColor, lowerRightRect, 0)
```

The first parameter to `pygame.draw.rect` specifies the surface on which the rectangle is to be drawn.

The second parameter specifies the color of the rectangle's border.

The third parameter specifies the location, width, and height of the rectangle.

The fourth parameter specifies the width of the border. If this value is set to zero, as is the case here, the rectangle is filled with the color specified by the second parameter.

Compute the alpha value for the circle

The code in [Listing 6](#) computes the alpha value that will be used to draw the circle during the next iteration of the runtime loop. *(The alpha value from the previous iteration is used to compute the **Color** object for the circle at the beginning of this iteration of the runtime loop in [Listing 3](#). I should have put the code in [Listing 6](#) at the top of the runtime loop just ahead of the code in [Listing 3](#) for better organization.)*

Listing 6. Compute the alpha value for the circle.

```
if alpha <= (255 - alphaInc):  
    alpha += alphaInc #Increase alpha  
else:  
    alpha = 0 #Reset alpha to zero
```

Create a new surface with a format that contains per pixel alphas

The statement in [Listing 7](#) calls the [convert_alpha](#) method of the [Surface](#) class on the **baseSurf** object to create a new surface with the same dimensions and a format that contains **per pixel alphas**. In other words, each pixel in the new surface format has its own alpha value.

Apparently the pixel alpha values are initially set to zero. Areas of the surface other than those on which the circle is drawn are transparent allowing the colored rectangles on the base surface to show through unchanged as shown in [Figure 1](#).

Listing 7. Create a new surface with a format that contains per pixel alphas.

Listing 7. Create a new surface with a format that contains per pixel alphas.

```
mask = baseSurf.convert_alpha()  
  
"""  
#This is another way to create a surface with per  
pixel alpha enabled.  
mask = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)  
"""
```

The code inside the comment in [Listing 7](#) shows another way to create a **Surface** object with a format that contains per pixel alphas. With this approach, the **Surface** class constructor is called to create a new surface with specified arbitrary dimensions and the flag [pygame.SRCALPHA](#). It is the flag that causes the surface format to contain per pixel alphas. Either approach will work in this program because the dimensions of the new surface need to match the dimensions of the base surface.

Draw a circle on the mask

The statement in [Listing 8](#) calls [pygame.draw.circle](#) to draw a circle on the mask passing an object and several variables that were created earlier as parameters..

Listing 8. Draw a circle on the mask.

```
pygame.draw.circle(mask, circleColor,  
(WIDTH//2, HEIGHT//2), RADIUS, BORDERWIDTH)
```

The first parameter to **pygame.draw.circle** specifies the surface on which the circle is to be drawn.

The second parameter specifies the color (*and transparency*) of the circle's border.

The third parameter specifies the coordinates of the center of the circle.

The fourth parameter specifies the radius of the circle.

The fifth parameter specifies the width of the border. If the border width is specified as zero, the circle will be filled with the border color and transparency. However, unlike with the rectangles drawn earlier, the border width for the circle is not set to zero for this program. This produces an output that might technically be referred to as an annulus.

The remainder of the code

The remainder of the code in the program is shown in [Listing 9](#)

Listing 9. The remainder of the code.

```
#Draw the mask on baseSurf.
baseSurf.blit(mask, (ORIGIN))

#Display baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame rate in
frames per second

pygame.quit() #Terminate the program outside the runtime
loop
```

I have explained code like that shown in [Listing 9](#) in earlier modules. Therefore, this code should not require an explanation beyond that provided by the comments.

Run the program

I encourage you to copy the code from [Listing 10](#). Execute the code and confirm that you get results similar to those shown in [Figure 1](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is provided in [Listing 10](#).

Listing 10. Complete program listing.

```
"""
File Transparency03.py
An animated demonstration of per pixel alpha blending with a
draw function.
=====
"""

import pygame #Import required library
pygame.init() #Initialize imported pygame modules

quit = False #Initialize termination control variable.
pygame.display.set_caption("Transparency03.py") #Set the
window caption
FRAMERATE = 20 #Used to control the maximum frame rate.
WIDTH = 200 #Width of output window
HEIGHT = 200 #Height of output window
RADIUS = WIDTH//2 #Radius of circle
BORDERWIDTH = RADIUS//2 #Width of circle border
ORIGIN = (0,0) #Location of the mask on the base surface
alpha = 0 #Initial opacity of circle
alphaInc = 1 #Variable used to change opacity of circle

#Create and set the display mode on the base surface.
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object to help track
time

#Create some objects of type Rect to use later with
pygame.draw.rect
upperLeftRect = pygame.Rect(0,0,WIDTH/2,HEIGHT/2)
upperRightRect = pygame.Rect(WIDTH/2,0,WIDTH/2,HEIGHT/2)
lowerRightRect =
pygame.Rect(WIDTH/2,HEIGHT/2,WIDTH/2,HEIGHT/2)

#Create some color objects
upperLeftColor = pygame.Color(0,0,255) #Upper left color
upperRightColor = pygame.Color(0,255,0) #Upper right corner

#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final iteration of the
runtime loop.
```

```

circleColor = pygame.Color(255,0,0,alpha) #Circle color

#Create a Color object that matches the color that results
from blending
# the circle color with the color of the upper right
rectangle
lowerRightColor = pygame.Color(
    int(circleColor.r*circleColor.a/255+
        upperRightColor.r*(1-circleColor.a/255)),
    int(circleColor.g*circleColor.a/255+
        upperRightColor.g*(1-circleColor.a/255)),
    int(circleColor.b*circleColor.a/255+
        upperRightColor.b*(1-circleColor.a/255)))

#Draw some filled rectangles on baseSurf
pygame.draw.rect(baseSurf,upperLeftColor,upperLeftRect,0)

pygame.draw.rect(baseSurf,upperRightColor,upperRightRect,0)

pygame.draw.rect(baseSurf,lowerRightColor,lowerRightRect,0)

#Compute alpha value for this iteration of the runtime
loop.
if alpha <= (255 - alphaInc):
    alpha += alphaInc #Increase alpha
else:
    alpha = 0 #Reset alpha to zero

#Create a new surface with per pixel alpha enabled.
Apparently pixel alpha
# values are initially zero.
mask = baseSurf.convert_alpha()

"""
#This is another way to create a surface with per pixel
alpha enabled.
mask = pygame.Surface((WIDTH,HEIGHT),pygame.SRCALPHA)
"""

#Draw a circle on the mask
pygame.draw.circle(mask,circleColor,
(WIDTH//2,HEIGHT//2),RADIUS,BORDERWIDTH)

#Draw the mask on baseSurf.

```

```
baseSurf.blit(mask, (ORIGIN))

#Display baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame rate in frames
per second

pygame.quit() #Terminate the program outside the runtime loop
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2235-Color-Part 4
- File: Itse1359-2235.htm
- Published: 01/31/16
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2240-Color-Animated Fly Through an RGB Color Cube

This module explains a program that provides an animated fly through an RGB color cube.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Red, green, and blue](#)
 - [Electromagnetic waves, visible light, frequency and wavelength](#).
 - [Cathode ray tubes](#)
 - [The organization of digital color data](#)
 - [An RGB color cube](#)
 - [An online ColorPicker](#)
 - [An animated fly through program](#)
 - [Access to individual pixels](#)
- [Discussion and sample code](#)
 - [The beginning of the program](#)
 - [Populate the pixel array](#)
 - [Decrement the blue value and repeat the process](#)
 - [The remaining program code](#)
 - [A final comment](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) An RGB color cube.
- [Figure 2.](#) Four slices through the color cube. .

Listings

- [Listing 1.](#) The beginning of the program.
- [Listing 2.](#) Populate the pixel array.
- [Listing 3.](#) Decrement the blue value and repeat the process.
- [Listing 4.](#) The remaining program code.
- [Listing 5.](#) Complete program listing.

General background information

Red, green, and blue

In an earlier module, I told you that there are at least **six different ways** that color can be represented in a **pygame** program:

1. **RGB and RGBA**
2. **CMY**
3. HSV and HSVA
4. HSL and HSLA
5. |1|2|3
6. #rrggb and #rrggbbaa

RGB and *RGBA* are the most fundamental representations. They are based on the properties of visible light, the physical characteristics of computer monitors, and the physical characteristics of computer memory. Virtually all modern computers support RGB.

CMY is based on the physical characteristics of printer ink.

HSV and HSL are based on the human perception of color as opposed to the physical properties of anything.

I'm not even going to try to describe item 5 in the [above list](#). If you want to learn more about it, go to Google and search for "*opponent color space representation of I1I2I3*".

Number 6 in the [above list](#) is essentially the same as number 1 but specified using a different syntax.

The letters in the acronym RGB represent the following color components (*often called channels in computer graphics lingo*) :

- R - red
- G - green
- B - blue
- A - alpha (*transparency*)

This module will concentrate on the RGB representation of color. I will explain the HSV representation in a future module.

Electromagnetic waves, visible light, frequency and wavelength

The universe is full of electromagnetic waves. Your cell phone uses relatively low frequency electromagnetic waves (*radio frequency waves*) to communicate with a cell phone tower. Your TV remote control uses electromagnetic waves in the *infra red* or IR frequency range to communicate with the television. Electromagnetic waves in the *ultra violet* or UV frequency range cause sunburns and skin cancer. Electromagnetic waves in the X-ray frequency range inform the dentist as to the internal state of your teeth and gums. The [SETI Institute](#) uses radio telescopes to search for coherent electromagnetic waves from space. Last but not least, electromagnetic waves in the "*visible light*" frequency range (*spectrum*) impinge upon the retina in your eyes to let you see the world around you.

Electromagnetic waves have a *frequency* which is measured in cycles per second. The reciprocal of frequency is *wavelength* measured in meters per cycle. A rough analogy to frequency and wavelength would be to stand on the beach and measure how frequently the waves come onto the shore (*frequency in cycles per second*) or to estimate the distance between the waves (*wavelength in meters per cycle*) .

In theory, pure sunlight contains about equal amounts of all wavelengths in the visible frequency spectrum. A prism can be used to separate those wavelengths onto different paths and to project them onto a screen. When this is done, the different wavelengths appear as different colors to a human observer. The waves with the longest wavelengths appear to humans as something that is identified in the English language as red. The shortest wavelengths appear as something that is identified in the English language as violet or magenta.

The wavelength of visible light is very small. The visible light spectrum extends from about 400 nanometers to about 750 nanometers. Within that range we find:

- red - longest wavelength
- orange
- yellow
- green
- cyan
- blue

- violet - shortest wavelength

You can view a webpage [here](#) that contains a chart showing the wavelengths of the different colors.

Cathode ray tubes

In the development of color television (*or perhaps color photography*) someone determined that most of the colors that are useful for human vision can be created by combining various amounts of red, green, and blue light. These have come to be known as the [additive primary colors](#) ([RGB](#)) as opposed to the [subtractive primary colors](#) (*cyan, magenta, and yellow or CMY*) .

The cathode ray tubes (*CRTs*) used in early color televisions and CRT computer monitors were constructed of glass and shaped something like a bell with both ends closed. The glass on the large end was transparent. The inside surface of the glass on the large end was coated with phosphorous that glowed or transmitted light when bombarded with electrons. An electron gun was positioned at the small end. Electronic signals were used to cause a stream of electrons to bombard the phosphorous on the inside of the tube (*similar to a stream of water from a hose washing a car*) .

For color television, the phosphorous coating actually consisted of three different types of phosphorous. One type would glow red. Another type would glow green, and the third type would glow blue. The phosphorous was arranged in tiny dots on the inside of the glass in groups of three dots. One dot would glow red, one dot would glow green, and the other dot would glow blue. By carefully aiming the electron stream at each group of dots is such a way as to control the intensity with which each dot glowed, the system was able to control the color of the light transmitted by each group of dots. In other words, by glowing, each group transmitted an additive mixture of red, green, and blue light.

The colors perceived by a human observer could be controlled by individually controlling the intensity of the light transmitted by each dot in

each group of three dots. Television signals were composed and transmitted in such a way as to consist of separate red, green, and blue signal channels. The red, green, and blue signals were used to control the electron gun in such a way as to cause the desired dots to glow and the desired colors to be produced on the television screen.

The organization of digital color data

This concept was carried forward when CRT color monitors were developed for use with computers. The organization of the digital data representing color in the computer was designed so that it could easily be converted into red, green, and blue signals by the electronic interface that connected the monitor to the computer. Originally, various complicated schemes were used to minimize the amount of memory required to store the data for a colored image.

This data organization has evolved over time as computer memory has become less expensive. As of 2015, most computer systems organize color data in terms of pixels (*picture elements*). Each pixel ultimately causes a colored dot to appear on the computer screen when the colored image is displayed by a program.

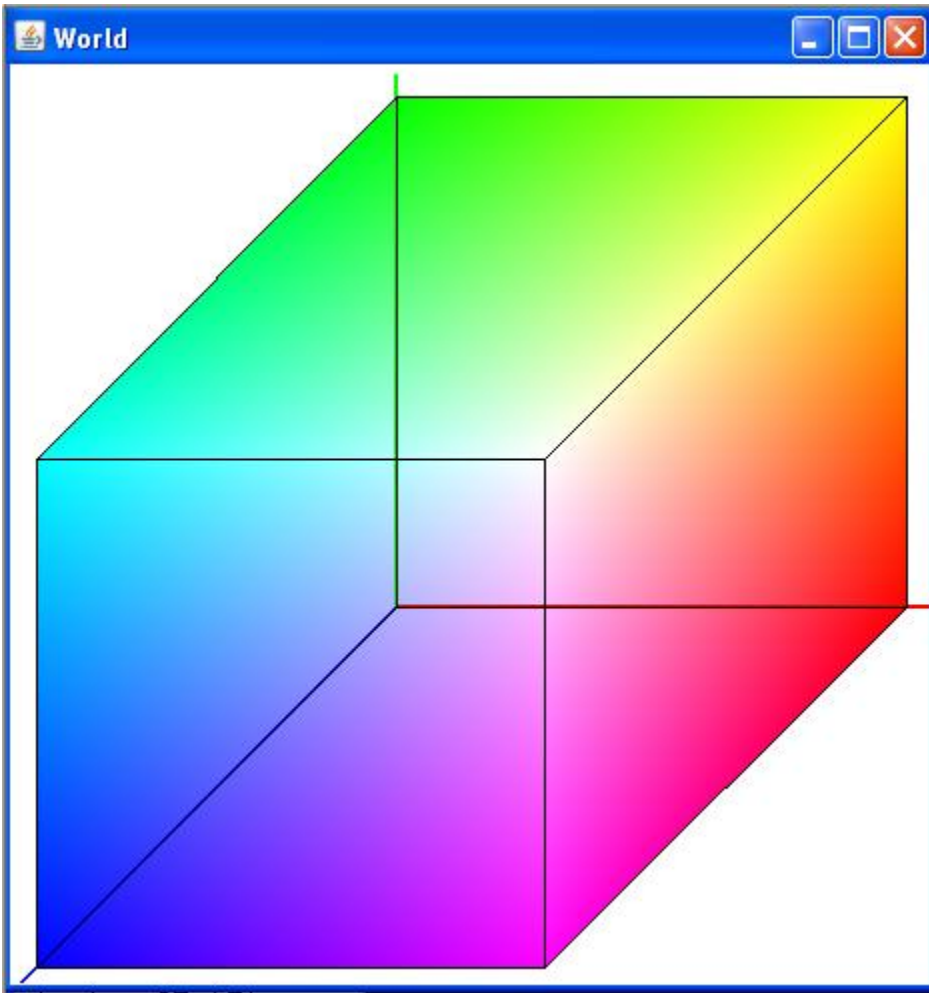
The digital data for each pixel usually consists of three (*and possibly four*) eight-bit bytes. One byte contains the data for the primary color red, one byte contains the data for the primary color green, and the third byte contains the data for the primary color blue. The fourth byte, if it exists, contains the data for transparency or opacity that you learned about in an earlier module. This is commonly known as 24-bit color with or without an alpha channel. Hence the names [RGB and RGBA](#).

Each eight-bit byte can contain the data for 256 different intensity levels of the primary color that it represents ranging from black to bright red, bright green, or bright blue for example. Given that there are three such 8-bit bytes in combination, the total number of combinations of bits is two to the 24th power, 256 to the third power, or 16,777,216. In theory a 24-bit color system can represent 16,777,216 different colors.

An RGB color cube

One way to envision the possible colors is to think of them as residing inside and on the surface of a cube, 256 units along each edge, as shown in [Figure 1](#).

Figure 1. An RGB color cube.



With the 24-bit RGB color system, the color black is represented by all three bytes having a value of zero. The color white is represented by all three bytes having a value of 255. The primary colors (*red, green, and blue*) are represented when one byte has a value of 255 and the other two bytes have a value of zero. The secondary colors (*yellow, cyan, and magenta*) are represented when two bytes have a value of 255 and the third byte has a value of zero.

Given the syntax (red,green,blue), the following colors are produced by the combinations of value shown:

1. black (0,0,0)
2. red (255,0,0)
3. yellow (255,255,0)
4. green (0,255,0)
5. cyan (0,255,255)
6. blue (0,0,255)
7. magenta (255,0,255)
8. white (255,255,255)

The cube shown in [Figure 1](#) is plotted against in a 3D coordinate system where the axes are red, green, and blue. The red axis is horizontal to the right. The green axis is vertical. The blue axis is intended to represent an axis protruding from the screen.

Although it might not look like it, the cube is intended to have equal length edges with each edge extending from 0 to 255. The coordinates of each of the eight vertices represents one of the eight combinations of color values given [above](#). The (*hidden*) coordinates of the origin represent black (red=0,green=0,blue=0).

The front, top, right vertex represents white (red=255,green=255,blue=255).

The front, bottom, right vertex represents magenta (red=255,green=0,blue=255), etc.

As you can see, except for the hidden vertex at the origin, the color of the image at each vertex in [Figure 1](#) matches the corresponding colors in the [above list](#).

Three of the surfaces of the cube are shown and the other three are hidden. The color shown at any point on any of the three visible surfaces is intended to represent the color produced by a 24-bit RGB color system value when the three color values are used as the coordinates of the point. (*Obviously the colors at points inside the cube and the colors of points on the hidden surfaces are not shown in [Figure 1](#).*)

An online ColorPicker

Given enough computer time and enough publication space, I could show all 16,777,216 colors by drawing 256 slices through the cube parallel to any of the faces on the cube. You probably aren't interested in seeing that and I'm not interested in producing it. However, you may be interested in experimenting with the online [ColorPicker](#) that lets you interactively fly through the cube by moving the interactive slider on the right to view the different color planes. Alternatively, you can enter RGB values or HSB values in the cells on the right to see the colors produced by those values.

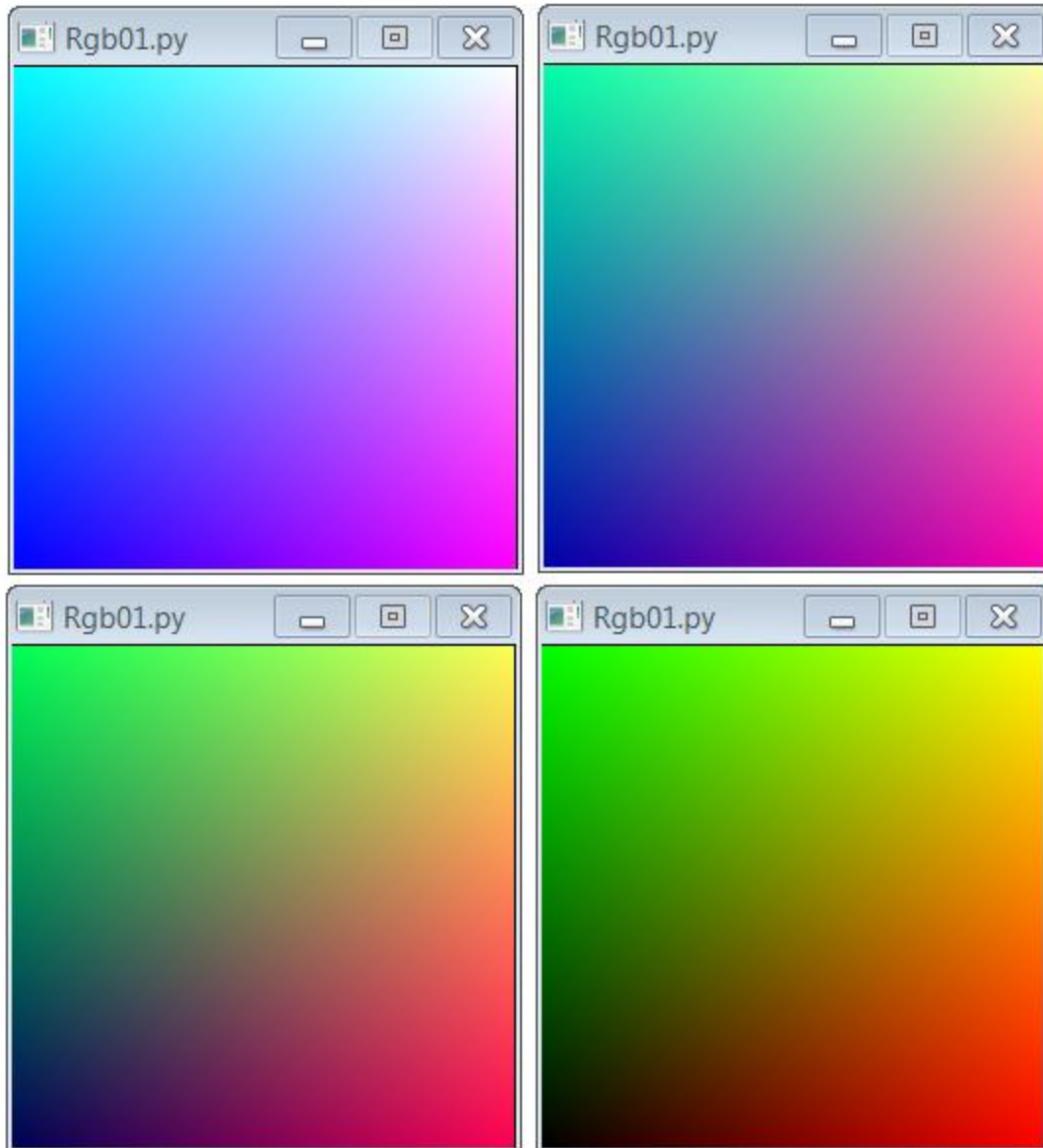
For example, the official RGB values for [burnt orange](#) at the University of Texas are (191,87,0). Try entering those values into the text fields for R, G, and B and note the color that appears above the text fields as well as the color underneath the small round cursor. *(If you happen to live in central Texas as I do, that color should be familiar to you as one of the official colors of the University of Texas. The other official color at UT is white.)*

The color picker also shows the HSB values for any given color. These values are essentially the same as the HSV values that I will discuss in a future module.

An animated fly through program

You may also be interested in the following animated **pygame** program. It begins at the front face of the cube in [Figure 1](#) and flies through the cube from front to back showing you the 65,536 colors on each of 255 slices that are parallel to the front face. The first slice for a blue value of 255 is shown in the upper left image in [Figure 2](#). Note the similarity between that slice and the front face of the cube in [Figure 1](#).

Figure 2. Four slices through the color cube.



The upper right image in [Figure 2](#) shows the slice for a blue value of 170. The bottom left image shows the slice for a blue value of 85, and the bottom right image shows the slice for a blue value of 0, which is the back face of the cube.

This program contains some features that are new to this module so may might want to pay particular attention to the code.

Access to individual pixels

Although the primary topic of this module continues to be about color, an important new concept is introduced here -- access to individual pixels. The code in this module, and several future modules, will use an object of the class [PixelArray](#) to wrap an object of type [Surface](#) to provide direct access to the surface's pixels. Several future modules will manipulate the color and the alpha values of individual pixels. Some of those modules will use a more sophisticated form on pixel access based on [pygame.surfarray](#) and a numeric library named [Numpy](#).

Discussion and sample code

A complete listing of the program is provided in [Listing 5](#) near the end of the module. I will explain the program in fragments.

The beginning of the program

The beginning of the program is shown in [Listing 1](#). With one exception, all of the code in [Listing 1](#) should be familiar to you by now.

Listing 1. The beginning of the program.

```
import pygame #Import required library
pygame.init() #Initialize imported pygame
modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Rgb01.py") #Set
the window caption
FRAMERATE = 40 #Used to control the maximum
```

Listing 1. The beginning of the program.

```
frame rate.
WIDTH = 256 #Width of output window
HEIGHT = 256 #Height of output window

#Initialize working variables
red = 0
redRange = 255
green = 0
greenRange = 255
blue = 255
blueRange = 255

#Create and set the display mode on the base
surface.
baseSurf =
pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object
to help track time

#Get an array that wraps the pixels on the
base surface making it possible
# to set the color of each pixel individually
based on its coordinate location.
pxarray = pygame.PixelArray(baseSurf)

#Create a color object for use in coloring the
pixels
color = pygame.Color(0,0,0)

#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.
```

Listing 1. The beginning of the program.

The exception is the statement that begins with **pxarray** , which creates an object of the class [PixelFormat](#) . An object of type **PixelFormat** wraps an object of type [Surface](#) and provides direct access to the surface's pixels.

A pixel array can be one or two dimensional. A two dimensional array, like its surface, is indexed [column, row]. (*I will use a two dimensional pixel array in this program.*) This object will be used once during each iteration of the runtime loop to set the color of each of the pixels belonging to the base surface referred to by **baseSurf** . The use of an object of type **PixelFormat** is new to this module but will be heavily used in future modules.

[Listing 1](#) also creates a [Color](#) object referred to by the variable named **color** . This object along with some of the working variables created in [Listing 1](#) will be used to set the color of each pixel inside the runtime loop.

Populate the pixel array

The code in [Listing 2](#) is inside the runtime loop. At this point, the blue color value [has been set](#) for this iteration. The code in [Listing 2](#) uses a pair of nested **for** loops to populate the 65,536 pixels in the base surface with colors derived from the current blue value in combination with all possible combinations of red and green values.

Listing 2. Populate the pixel array.

Listing 2. Populate the pixel array.

```
#Populate the pixel array from left to
right, top to bottom.
for red in range(redRange):
    for green in range(greenRange):
        color.r = red
        color.g = green
        color.b = blue
        pxarray[red,255-green] = color
```

For a blue value of 170, for example, this produces the 65,536 different colors shown in the top right image in [Figure 2](#).

Decrement the blue value and repeat the process

In order to represent a fly through from the front face to the back face of the cube in [Figure 1](#), the program begins with a blue value of 255 and reduces it by one during each iteration of the runtime loop. When it reaches -1, it is reset to 255 and the cycle repeats. This is accomplished by the code in [Listing 3](#).

Listing 3. Decrement the blue value and repeat the process.

Listing 3. Decrement the blue value and repeat the process.

```
if blue > 0:  
    blue -= 1  
else:  
    blue = 255
```

The remaining program code

The remaining program code is shown in [Listing 4](#). There is nothing new here so further explanation should not be needed.

Listing 4. The remaining program code.

```
#Display baseSurf  
pygame.display.flip()  
  
clock.tick(FRAMERATE) #Control the frame  
rate in frames per second  
  
pygame.quit() #Terminate the program outside  
the runtime loop
```

A final comment

Although the RGB system for managing color data is mathematically and arithmetically straightforward, and reasonably efficient computationally, not everyone is happy with it. In particular, many people with an artistic bent say that it does not represent color in the same way that people perceive color.

Numerous attempts have been made to resolve that issue by defining alternative way to manage digital color data. Two of those alternative approaches are supported by **pygame** : HSV and HSL. You will probably need some understanding of alternative approaches in addition to RGB, so I will explain HSV in a future module.

Run the program

I encourage you to copy the code from [Listing 5](#). Execute the code and confirm that you get results similar to those shown in [Figure 2](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do. For example, you might want to hold the red value constant during each iteration and iterate on the blue and green values instead. That would cause you to fly through the cube beginning at a different face.

Complete program listing

A complete listing of the program is provided in [Listing 5](#).

Listing 5. Complete program listing.

```
"""
File Rgb01.py
This program animates a fly through an RGB color
cube displaying each of
256 slices that are parallel to the red-green
```

plane. The first slice is at a blue value of 255 showing the blue, cyan, white, and magenta vertices. The last slice is at a blue value of 0 showing the black, green, yellow, and red vertices.

After displaying the last slice, the blue value is reset to 255 and the cycle continues.

```
=====
=====
"""
```

```
import pygame #Import required library
pygame.init() #Initialize imported pygame modules
```

```
quit = False #Initialize termination control
variable.
```

```
pygame.display.set_caption("Rgb01.py") #Set the
window caption
```

```
FRAMERATE = 40 #Used to control the maximum frame
rate.
```

```
WIDTH = 256 #Width of output window
```

```
HEIGHT = 256 #Height of output window
```

```
#Initialize working variables
```

```
red = 0
```

```
redRange = 255
```

```
green = 0
```

```
greenRange = 255
```

```
blue = 255
```

```
blueRange = 255
```

```
#Create and set the display mode on the base
surface.
```

```
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
```

```
clock = pygame.time.Clock() #Create an object to
```

help track time

```
#Get an array that wraps the pixels on the base
surface making it possible
# to set the color of each pixel individually
based on its coordinate location.
pxarray = pygame.PixelArray(baseSurf)
```

```
#Create a color object for use in coloring the
pixels
color = pygame.Color(0,0,0)
```

```
#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.
```

```
    #Populate the pixel array from left to right,
top to bottom.
    for red in range(redRange):
        for green in range(greenRange):
            color.r = red
            color.g = green
            color.b = blue
            pxarray[red,255-green] = color
```

```
    #Decrement the blue color value by one unit
during each iteration of
    # the runtime loop and reset to 255 when it
reaches -1.
    if blue > 0:
        blue -= 1
    else:
        blue = 255
```



```
#Display baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame rate
in frames per second

pygame.quit() #Terminate the program outside the
runtime loop
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2240-Color-Part 5
- File: Itse1359-2240.htm
- Published: 01/08/15
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a

book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2245-Color-Explanation of the HSV Color Space
This module explains the HSV color space.

Table of contents

- [Preface](#)
 - [What you have learned](#)
 - [What you will learn](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Discussion](#)
 - [The RGB color space](#)
 - [The HSV color space](#)
 - [Specifying a color in the RGB color space](#)
 - [Specifying a color in the HSV color space](#)
 - [A description of Hue](#)
 - [A description of Saturation](#)
 - [A description of Value or Brightness](#)
 - [The cylindrical model](#)
 - [The available colors](#)
 - [Hue angles and associated names](#)
 - [Determining the Hue of a color](#)
 - [Determining the Saturation of a color](#)
 - [Let's reiterate](#)
 - [Let's talk some more about Value](#)
 - [A vertical slice through the center](#)
 - [Experiment with a color picker](#)
 - [The main elements of the color picker](#)
 - [Instructions for using the color picker](#)

- [Interesting experiments](#)
 - [Hold Hue and Saturation constant and vary Brightness](#)
 - [Hold Hue and Brightness constant and vary Saturation](#)
 - [Hold Saturation and Brightness constant and vary Hue](#)
- [The HSL color space](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

What you have learned

You have learned quite a lot so far including how to visualize the RGB color space as a 3D cube positioned at the origin of 3D Red, Green, and Blue axes .

What you will learn

Unlike most modules in this collection, this module won't explain program code. Instead, it will concentrate on concepts involving the RGB and HSV color spaces. Program code involving the HSV color space will be explained in a future module.

You will learn about the HSV color space along with learning about the relationship between the RGB color space and the HSV color space. You will learn also how to visualize the HSV color space as a 3D cylinder.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Top surface of an HSV color cylinder.
- [Figure 2.](#) HSV cylinder surface at $V = 60$.
- [Figure 3.](#) HSV cylinder surface at $V = 30$.
- [Figure 4.](#) Exposed face from a vertical slice through the center.
- [Figure 5.](#) An online ColorPicker.

Discussion

In an earlier module, you learned about the RGB color space and learned how to visualize that color space using a 3D color cube.

Near the end of that module, I told you that although the RGB system for managing color data is mathematically and arithmetically straightforward, and is reasonably efficient computationally, not everyone is happy with the RGB color space. In particular, many people with an artistic bent say that it is not intuitive and that it does not represent color in the same way that people perceive color.

Numerous attempts have been made to resolve that issue by defining alternative ways to manage and manipulate digital color data. Two of those alternative approaches are supported by **pygame** : HSV and HSL. I will explain the HSV color space in this module and show how you can visualize it as a 3D cylinder.

Before getting into the details, I recommend that you also study the material on the following two websites concurrently with your study of this module:

- [Primary Colors of Light and Pigment](#)

- [Hue, Value, Saturation](#)

The RGB color space

Particularly interesting is the last sentence in the following paragraph that was taken from [Primary Colors of Light and Pigment](#) :

"Different wavelengths of light are perceived as different colors. For example, light with a wavelength of about 400 nm is seen as violet, and light with a wavelength of about 700 nm is seen as red. However, it is not typical to see light of a single wavelength. *You are able to perceive all colors because there are three sets of cones in your eyes - one set that is most sensitive to red light, another that is most sensitive to green light, and a third that is most sensitive to blue light.* "

This physical characteristic of the human eye links us directly back to the RGB color space, and [Primary Colors of Light and Pigment](#) should help to reinforce what you learned in the earlier module regarding the RGB color space.

The HSV color space

In addition to the HSV color space, there is also a color space called the **HSB** color space that substitutes the word *Brightness* for the word *Value* . As near as I can tell, the two are mathematically identical. I prefer the word *Brightness* over the word *Value* because I believe it better describes its purpose. Also the word *value* is a common word in computer programming and multiple usage of the word for different purposes can lead to confusion, particularly when trying to write about two different meanings of the same word in the same article. **Pygame** purports to support HSV and does not purport to support HSB. On the other hand, we will be using a color picker

in this module that uses HSB in place of HSV. Given all of the above, I will use the terms HSV and HSB interchangeably in this module.

I couldn't possibly do a better job of introducing the various technical aspects of HSV than that provided by the author in [Hue, Value, Saturation](#), so I won't even try. However, I will depart from that explanation in one minor respect. The author of that article describes the HSV color space as a cone. I will describe it as a cylinder instead of a cone. The end result is the same but I believe that cylinder is the better description. (*A cylinder is also easier to model in a computer program.*)

Specifying a color in the RGB color space

You specify a color in the RGB color space by specifying three values: Red, Green, and Blue. As you learned in an earlier module, you can think of the three values as the coordinates of a point in a 3D space. You can also think of that point as a point on the surface or inside of a 3D cube with dimensions of 256 units along each edge.

All three values use the same scale. Depending on the software you are using, the range of the values will typically be expressed as:

- 0 to 255 for computer geeks like myself.
- 0 to 100 percent for those who like to work in percentages.
- 0.0 to 1.0 for those who like to work with decimal values.

Ultimately it doesn't matter which scale is used as long as the scale is used consistently and it is supported by the software being used.

Specifying a color in the HSV color space

Colors in the HSV color space are also specified by three values but they don't all have the same units. In the HSV color space:

- H is for Hue: 0 to 359 degrees

- S is for Saturation: 0 to 100 percent
- V is for Value: 0 to 100 percent

As I indicated earlier, some people believe that these three values are more intuitive and also do a better job of specifying color in the way that people perceive color. There are pros and cons to this. For example, just about everyone knows what I am talking about when I use the words Red, Green, and Blue. However, without the kind of training that you are receiving here, very few people without artistic training know what I am talking about when I use the words Hue and Saturation. The same is true for the use of the word Value in regards to color.

A description of Hue

The author of [Hue, Value, Saturation](#) provides a description of Hue and I encourage you to read it. I will also provide a description in my own words in hopes that will help you wrap your mind around this concept.

In my words, Hue is a color (*expressed as an angle*) that can be created by adding no more than two of the additive primary colors, Red, Green, and Blue in some amounts. The value of at least one of the primary colors will be zero for any value of Hue.

The colors at some Hue values have names like :

- Red - 0 degrees - 3 o'clock (*also 360 degrees on some systems*)
- Yellow - 60 degrees - 5 o'clock
- Green - 120 degrees - 7 o'clock
- Cyan - 180 degrees - 9 o'clock
- Blue - 240 degrees - 11 o'clock
- Magenta - 300 degrees - 1 o'clock

The colors at other Hue values may also have names such as some of the names listed [here](#).

Hue is specified as an angle between 0 and 359 degrees and is not specified as the name of a color. However, if you are developing an HSV color from

scratch, it helps to know how those angle values relate to color. The names of the colors at equal Hue intervals of 60 degrees beginning with zero are given in the [above](#) list. From that, you might surmise, for example, that the color orange has an angle somewhere between 0 degrees (*Red*) and 60 degrees (*Yellow*) .

A description of Saturation

I will also provide a description of Saturation in my own words. If a color contains only one or two of the additive primary colors and the other primary color(s) has a value of zero, it is fully saturated. (*It has a Saturation value of 100.*) If the color contains equal amounts of all three additive primary colors, it is totally unsaturated and has a Saturation value of zero. In that case, it will likely be perceived by a human observer as white, some shade of gray, or black.

If the color contains a mixture of all three primary colors in any amounts, The saturation value will be less than 100.

A description of Value or Brightness

In general, given any color, the color can be made darker or brighter by multiplying the Red, Green, and Blue color values by the same multiplicative factor provided that you don't cause the value of any color value to exceed 255. If you multiply by a factor less than 1.0, you decrease the *Value* or the *Brightness* . If you multiply by a factor greater than 1.0, you increase the Value or Brightness.

For example, if any color component has a value of 255, the Value or Brightness is 100. If all color components have a value of 0, the Value or Brightness is 0. If the largest color component has a value of 128, the Value or Brightness is 50.

Thus, the Value or Brightness of a color can range from 0 (*black*) at the low end to 100 (*brightest*) at the high end. (*White is brighter, has a greater*

Value, than black.)

The cylindrical model

The available colors

Now I am going to introduce you to a cylindrical model that can be used to envision HSV or HSB. Assume that you are looking down on the top of an HSV cylinder. [Figure 1](#) shows what you would see. It is the top surface of an HSV color cylinder.

Figure 1. Top surface of an HSV color cylinder.



What you see in [Figure 1](#) are all of the colors that are available for Value or Brightness equal to 100. You have access to all the colors shown in [Figure 1](#) plus 100 darker versions of those colors.

Hue angles and associated names

Now assume that you push a pin into the center of the circle in [Figure 1](#) and attach a very thin, very straight, and very strong wire to the pin so that it rotates around the center like the hour hand on a clock.

Assume that you position the wire so that it extends from the center to the Red edge on the right at 3 o'clock. That would represent a Hue of zero degrees. If you rotate the wire clockwise so as to align it with the Yellow (5 o'clock) , that would represent a Hue of 60 degrees. The color of the Hue for a particular Hue angle from zero to 359 degrees is the color under the wire where the wire intersects the edge of the circle. The angles and names for six well-known Hue values were listed [above](#).

All of the colors around the edge of the circle can be created through the addition of only two of the additive primary colors in differing amounts. *(For 0, 120, and 240 degrees, only one primary color has a non-zero value.)* For the top surface with a Value of 100 shown in [Figure 1](#), the color combination for every Hue angle has one primary color with a value of 255.

The named primary and secondary Hue colors occur every 60 degrees. The other Hue colors fall in between those angles.

Determining the Hue of a color

If you pick a point anywhere on the surface shown in [Figure 1](#), the Hue of the color can be determined by rotating the wire so that it intersects that point and then reading the angle of the wire clockwise relative to 3 o'clock. The Hue values range from 0 to 359 degrees. Clockwise rotation results in increasing positive Hue angles. *(An angle of 360 degrees is the same as an angle of zero degrees.)*

Determining the Saturation of a color

Now that you understand Hue, let's discuss Saturation. Saturation is the distance from the center of the circle to a point somewhere inside of or on the edge of the circle shown in [Figure 1](#). Maximum Saturation of 100 occurs at the edge of the circle. Therefore, the distance from the center to any point in the circle would be between 0 and 100. That distance would be the saturation of the color represented by that point.

For example, for a Hue of 90, I would describe the color with a Saturation of 50 to be a yellowish green in [Figure 1](#). The point representing that color lies half way between the center and the edge of [Figure 1](#) at the 6 o'clock angle.

Let's reiterate

[Figure 1](#) shows the top surface for an HSV cylinder for a Value of 100. You can get the Hue of a color by reading the angle from the 3 o'clock position to a line from the center intersecting the point representing the color. You can get the Saturation by measuring the distance from the center to the point. Given those two values along with a known Value of 100, you have everything you need to describe the color at the point on the top surface in HSV color space. *(You can read more about this in one of my earlier publications [here](#).)*

All of the colors at the edge of the circle are created by adding no more than two of the additive primary colors: Red, Green, and Blue. All of the colors on the edge are fully saturated. *(The value of Saturation at the edge is 100.)* The white dot at the center of the circle consists of the addition of maximum values of all three primary colors. The color white is totally unsaturated. *(The value of Saturation at the center is 0.)* All of the other colors on the surface are partially saturated and can contain a mixture of all three primary colors.

Let's talk some more about Value

The top surface of the cylinder shown in [Figure 1](#) represents a Value of 100. The bottom surface represents a Value of zero. *(It is totally black, so I won't bother showing it.)* If we were to slice through the cylinder 60-percent of the way up from the bottom, we would expose a surface with a Value of 60. That surface is shown in [Figure 2](#).

Figure 2. HSV cylinder surface at $V = 60$.



If you select a point at the same location in [Figure 2](#) as in [Figure 1](#), it would have the same Hue and the same Saturation, but it would have a Value of 60 instead of a Value of 100. It would simply be a darker version of the color that you selected in [Figure 1](#). *(Now you know one of the reasons that I prefer the term Brightness over the term Value. It is more descriptive of the physical process.)*

Let's look at one more slice through the cylinder at a Value of 30 as shown in [Figure 3](#). As you can see, the colors here are well on their way to becoming so dark as to be black.

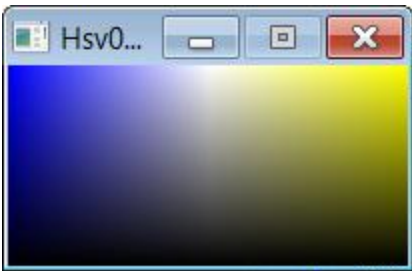
Figure 3. HSV cylinder surface at $V = 30$.



A vertical slice through the center

Now let's slice our cylinder one more time in a different way. Assume that we slice straight down through the cylinder along a line at 60 degrees, separating the cylinder into two halves, and then look at the exposed face of one half of the cylinder from a position that previously was the Green area in [Figure 1](#) (120 degrees or 7 o'clock) . That line would go through Yellow on the lower right and would go through Blue on the upper left in [Figure 1](#). The exposed face would look something like [Figure 4](#).

Figure 4. Exposed face from a vertical slice through the center.



The top edge of the image in [Figure 4](#) represents the top of the cylinder and the bottom edge represents the bottom of the cylinder. A long thin vertical spike driven straight down at the center of [Figure 1](#) would represent a vertical center line and would separate the image in [Figure 4](#) into two halves. That vertical center line would go through white at the top, gray in the middle and black at the bottom.

The Yellow color at the upper right corner of [Figure 4](#) is the same as the color at the edge of the circle at 60 degrees in [Figure 1](#). The Blue color at the upper left corner in [Figure 4](#) is the same as the color at the edge of the circle at 240 degrees in [Figure 1](#).

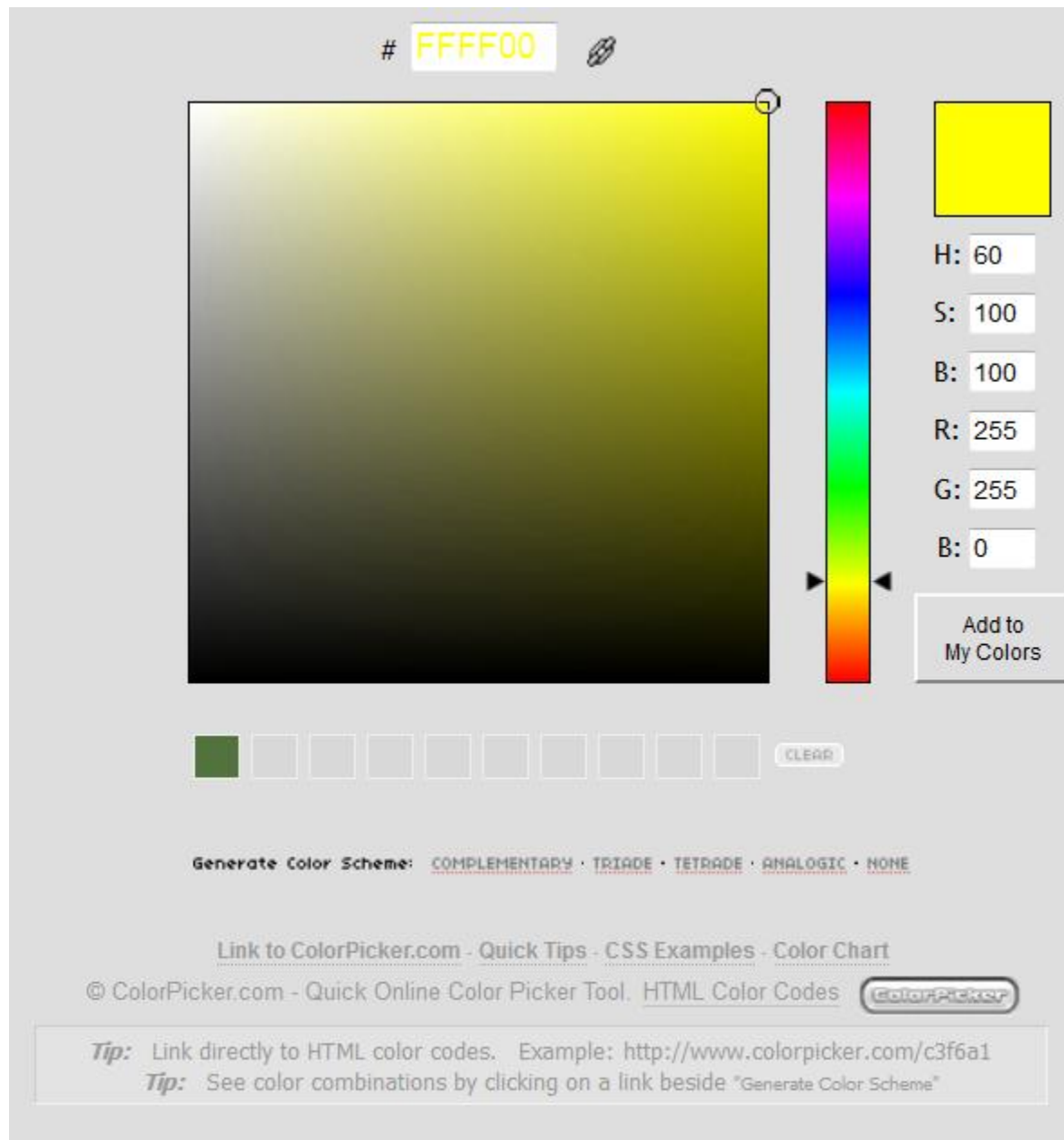
A point somewhere in the right half of [Figure 4](#) would have a Hue of 60 degrees, a Saturation equal to the absolute (*unsigned*) distance from the vertical center line of [Figure 4](#), and a Value or Brightness equal to distance from the bottom of Figure 4.

Similarly, a point somewhere in the left half of [Figure 4](#) would have a Hue of 240 degrees, a Saturation equal to the absolute distance from the vertical center line, and a Value equal to the distance from the bottom.

Experiment with a color picker

Let's approach this from another viewpoint. Open the [ColorPicker](#) web page in your browser. Unless the content of the page has changed, you should see the online color picker shown in [Figure 5](#). (Note that the colors may be different from those shown in [Figure 5](#) when the page opens in your browser.)

Figure 5. An online ColorPicker.



The main elements of the color picker

There are four main elements to the color picker:

- A large colored square on the left that has a small circular cursor that can be used to select a color within the square.
- A multi-colored vertical bar on the right with a slider indicated by pointers on each side of the bar.
- A small square in the upper right that shows the color selected by the cursor in the large square.
- A group of six text boxes labeled
 - H:
 - S:
 - B:
 - R:
 - G:
 - B:

The top three text boxes are intended to represent the H, S, and B values in an HSB color space (*as opposed to an HSV color space*) .

The bottom three text boxes are intended to represent the R, G, and B values in an RGB color space.

Instructions for using the color picker

Enter 60 into the topmost text box for Hue. Drag the cursor to the upper-right corner of the large square as shown in [Figure 5](#) or enter a 0 in the text box labeled S for Saturation and enter 100 in the text box labeled B for Brightness.

Note that there are two text boxes labeled B. One stands for Brightness, as in HSB and the other stands for Blue as in RGB. (*Maybe that is why some people prefer HSV over HSB - to avoid confusion between the two uses of the letter B.*)

Now compare the large square with the right half of [Figure 4](#). They should be the same except that the version in [Figure 5](#) is much larger. This tells us that the large square in the color picker represents the right half of a vertical slice in an HSB cylinder along a line given by a specific Hue in degrees.

Now enter 240 in the text box for Hue. What you see should be a mirror image of the left half of [Figure 4](#).

You may have noticed that the slider to the left of the text boxes moved when you entered a value in the text box for Hue. You can also move that slider up and down manually. Moving it up and down is equivalent to rotating my hypothetical wire in [Figure 1](#) and then slicing down through the cylinder along the wire to expose the face.

The colored vertical bar next to the text boxes in the color picker is analogous to the edge of the circle in Figure 1. You can select a Hue by moving the slider up and down. The lowest position of the slider represents a Hue of 0 degrees. The highest position of the square represents 360 degrees, which is the same as 0 degrees. If you manually drag the slider to the top, a 0 appears in the Hue box and the slider stays at the top. However, if you enter 360 in the Hue box, the slider automatically moves to the bottom and the value in the Hue box changes to 0.

Once you select a Hue, you can move the cursor horizontally to select a value for Saturation. You can move the cursor up and down to select a value for Brightness.

As you move the slider up and down and/or move the cursor back and forth and up and down, the numbers in the text boxes change to indicate the HSB values and the RGB values for the color specified by the combined positions of the cursor and the slider.

Interesting experiments

There are some interesting experiments that you can perform with the color picker to get a better feel for the complex relationship between HSV and RGB. We will begin with the one that is the most straightforward.

Hold Hue and Saturation constant and vary Brightness

Pick any value for Hue, 199 for example, and enter it into the Hue box. Pick any value for Saturation, 60 for example, and enter it into the Saturation box. Enter 100 in the Brightness box. The cursor should be at the top edge of the large square.

Note the values for R, G, and B. Now drag the cursor straight down the screen trying to maintain the Saturation value constant. The values for Brightness, Red, Green, and Blue should all decrease linearly and should be zero when the cursor reaches the bottom.

Hold Hue and Brightness constant and vary Saturation

Pick any value for Hue, 199 for example, and enter it into the Hue box. Pick any value for Brightness, 50 for example, and enter it into the Brightness box. Enter 100 into the Saturation box and note the values for Red, Green, and Blue. One of the values for RGB will be larger than the other two. (*If you chose 60, 180, or 300 for Hue, two of the RGB colors will have the same value.*) Now drag the cursor from right to left being careful to hold the value for Brightness constant. The large value that you noted before you started dragging should not change, but the other two values should increase until all three values are equal when the cursor reaches the left side with a Saturation value of 0. In this case, the change in value is linear as the cursor moves from right to left.

Hold Saturation and Brightness constant and vary Hue

Pick any value for Saturation, 60 for example, and enter it into the Saturation box. Pick any value for Brightness, 30, for example, and enter it into the Brightness box. Pick any Hue value between 0 and 359, 199 for example, and enter it into the Hue box. Note the RGB values.

One of the values for RGB will be larger than the other two. (*If you choose 60, 180, or 300 for Hue, two of the RGB colors will have the same value.*) Now drag the Hue slider up and down. Regardless of the position of the

slider, one and possibly two of the RGB values will continue to have that same maximum value. Among Red, Green, and Blue, the maximum value switches from one color to the next as the Hue crosses the 30, 180, and 300 degree marks.

Given the behavior of the RGB values in this third experiment, you may also conclude that thinking of color in terms of HSB or HSV is more intuitive than thinking of color in terms of RGB, even if you don't have an artistic bent. Just remember that most computers don't natively deal with colors according to an HSV color space. Digital color data is stored and manipulated in most computers in an RGB format. Working with HSV or HSB requires that the values be converted back and forth between RGB and HSV. This, in turn, frequently requires more programming effort. Fortunately, the extra programming effort required in **pygame** is minimal.

The HSL color space

I don't plan to discuss HSL. However, there are some interesting HSL color pickers in the following list. You may find it instructive to experiment with them.

- [HSL Color Picker - by Brandon Mathis](#)
- [HSL Color Picker; HTML Color Picker; Web, HEX, CSS, HSLa](#)
- [Colorizer - A color picker and converter \(RGB HSL HSBHSV CMYK HEX\)](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2245-Color-Part 6
- File: Itse1359-2245.htm

- Published: 01/10/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2250-Color-An Animated Dive Into an HSV Color Cylinder
This module demonstrates an animated dive down through the center of an HSV color cylinder.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
- [Run the program](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Six screen shots taken at random.

Listings

- [Listing 1](#). The preliminaries.
- [Listing 2](#). Iterate across all coordinates on the surface.
- [Listing 3](#). Test for inside the circle and set the Saturation if inside.
- [Listing 4](#). Compute the Hue angle.
- [Listing 5](#). Use HSVA parameters to set the color of a Color object.
- [Listing 6](#). Set the color of the current pixel.
- [Listing 7](#). Decrease or reset Value for HSV.
- [Listing 8](#). The remainder of the program code.
- [Listing 9](#). Complete program listing.

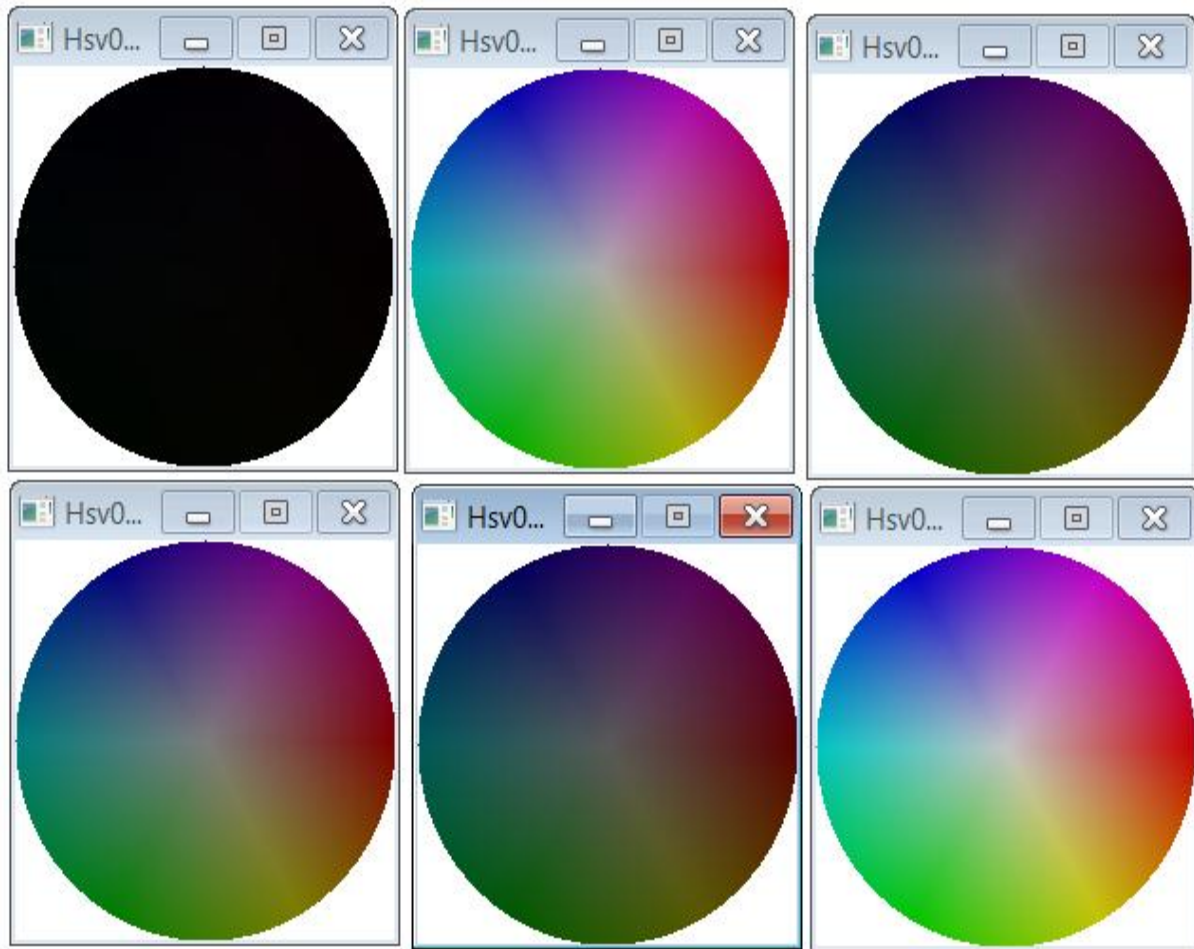
General background information

In an earlier module, you learned about the *HSV color space* and how that color space can be envisioned as a 3D cylinder with a radius of 100 units and a height of 100 units.

You will learn to work with HSV color using the **pygame** library in this module. I will explain an animated **pygame** program that dives into the top of an HSV color cylinder and displays the colors at each of 100 horizontal slices through the cylinder. When the animation reaches the bottom (*black*) surface, it will reset to the top of the cylinder and the cycle will continue.

[Figure 1](#) shows six screen shots taken at random while the program was running. Each of the circles represents a horizontal slice through the HSV color cylinder. The colors in each circle show the colors that reside in the cylinder at the vertical location of the slice. The slice in the upper-left corner of [Figure 1](#) is near the bottom (*black*) surface of the cylinder. The slice in the lower-right corner is near the top of the cylinder.

Figure 1. Six screen shots taken at random.



Discussion and sample code

I will explain the code for this program in fragments. The first fragment is provided in [Listing 1](#)

Listing 1. The preliminaries.

```
import math
import pygame #Import required library
```

Listing 1. The preliminaries.

```
pygame.init() #Initialize imported pygame
modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Hsv01.py") #Set
the window caption
FRAMERATE = 10 #Used to control the maximum
frame rate.
WIDTH = 200 #Width of output window
HEIGHT = 200 #Height of output window

#Initialize HSVA parameters
hue = 0 #Initial hue value
saturation = 0 #Initial saturation value
value = 100 #Start at the top and animate
downward.
valueRange = 100 #Height of cylinder
radius = 100 #Radius of the cylinder
radiusSq = radius ** 2 #squared
ALPHA = 100 #A constant in this program. Could
be omitted.

#Create and set the display mode on the base
surface.
baseSurf =
pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object
to help track time

#Get an array that wraps the pixels on the
base surface making it possible
# to set the color of each pixel individually
based on its coordinate location.
pxarray = pygame.PixelArray(baseSurf)
```


Listing 1. The preliminaries.

```
#Create a color object for use in transforming
from HSVA to RGB
color = pygame.Color(0,0,0)

#Create a Color object for the background of
the surface.
white = pygame.Color(255,255,255)

#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.

    baseSurf.fill(white)
```

Code similar to all of the code shown in [Listing 1](#) has been explained in earlier modules. Therefore, no explanation of that code beyond the embedded comments should be required in this module.

Iterate across all coordinates on the surface

The code in [Listing 2](#) shows the use of a pair of nested **for** loops to examine every pixel on the surface.

Listing 2. Iterate across all coordinates on the surface.

```
for xCnt in range(WIDTH):  
    for yCnt in range(HEIGHT):
```

The coordinates of each pixel will be examined to determine if that pixel falls inside or outside of the circle as shown in [Figure 1](#). If the pixel is outside the circle, the pixel is ignored and its color remains white as established by the call to **fill** in [Listing 1](#). If the pixel is inside the circle, the color of the pixel is modified to represent the HSV color at that location inside the cylinder.

Test for inside the circle

The code in [Listing 3](#) performs the test described above and sets the Saturation of the pixel is inside the circle.

Listing 3. Test for inside the circle and set the Saturation if inside.

Listing 3. Test for inside the circle and set the Saturation if inside.

```
xCntSq = (xCnt-WIDTH//2)**2
yCntSq = (yCnt-HEIGHT//2)**2

if xCntSq + yCntSq <= radiusSq:
    #The coordinate is inside the
circle. Compute and display
    # a color for it.
    saturation = (xCntSq + yCntSq)
** 0.5 #square root
```

The test is performed by computing the square of the distance from the pixel to the center of the circle and comparing that distance to the square of the radius of the circle. The Saturation is the distance from the pixel to the center of the circle. When the pixel is inside the circle, the Saturation is set as the square root of the square of the distance of the pixel from the center of the circle.

Compute the Hue angle

[Listing 4](#) uses Python's [math.atan2](#) function to compute the Hue angle.

Listing 4. Compute the Hue angle.

Listing 4. Compute the Hue angle.

```
        angInRad = math.atan2(-(yCnt -  
radius), -(xCnt - radius)) + math.pi  
        angInDeg = angInRad *  
180/math.pi  
  
        if angInDeg > 359:  
            angInDeg = 0
```

The code in [Listing 4](#) changes the sign of the x and y coordinates to place the red color at 3 o'clock, to place the green color at 7 o'clock, and to place the blue color at 11 o'clock. Without those changes, the red color would be at 9 o'clock.

The **math.atan2** function returns the angle in radians in the range from -[math.pi](#) to + [math.pi](#). The code in [Listing 4](#) adds **math.pi** to the returned value to cause **angInRad** to represent the angle in the range from 0 to **2*math.pi**.

Then the code in [Listing 4](#) converts the angle from radians to degrees and stores it in the variable named **angInDeg**.

The Hue angle needs to be in the range from 0 to 359 degrees. The computation described above can return a value of 360.0 degrees, which is the same as zero degrees. The code in [Listing 4](#) converts any angle value greater than 359 degrees to zero degrees.

Use HSVA parameters to set the color of a Color object

The code in [Listing 1](#) creates a black **Color** object and stores that object's reference in the variable named **color**.

The code in [Listing 5](#) changes the color stored in that object by setting the [hsva attribute](#) to a [List](#) containing **angInDeg** , **saturation** , **value** , and **ALPHA** .

Listing 5. Use HSVA parameters to set the color of a Color object.

```
color.hsva =  
[angInDeg, saturation, value, ALPHA]
```

The initial value for **value** is set to 100 in [Listing 1](#). It will be [modified](#) during each iteration of the runtime loop. The initial value for **ALPHA** is also set to 100 in [Listing 1](#) and does not change. It could be omitted from the code and the results would be the same. The base surface does not support alpha transparency.

Set the color of the current pixel

The code in [Listing 6](#) sets the color of the pixel at the current **xCnt** , **yCnt** location to the color that is now stored in the **Color** object referred to by **color** , using the **PixelArray** object that was created in [Listing 1](#). (You learned about *PixelArray* objects in an earlier module) .

Listing 6. Set the color of the current pixel.

```
pxarray[xCnt, yCnt] = color
```

Decrease or reset Value for HSV

The iteration process that began in [Listing 2](#) continues until the color of every pixel in the circle for the current **value** has been set. Then the code in [Listing 7](#) is executed to either decrease **value** or to reset it to 100. The cycle repeats until the user terminates the program.

Listing 7. Decrease or reset Value for HSV.

```
value -= 1  
if value < 0:  
    value = valueRange
```

The remainder of the program code

The remainder of the program code is shown in [Listing 8](#). There is nothing new about this code so it shouldn't require further explanation.

Listing 8. The remainder of the program code.

```
#Display baseSurf
pygame.display.flip()

clock.tick(FRAMERATE) #Control the frame
rate in frames per second

pygame.quit() #Terminate the program outside
the runtime loop
```

Run the program

I encourage you to copy the code from [Listing 9](#). Execute the code and confirm that you get results similar to those shown in [Figure 1](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listings

A complete listing of the program is provided in [Listing 9](#).

Listing 9. Complete program listing.

```
"""
File Hsv01.py
This animated program dives into the top of an
HSVA cylinder and displays the
surfaces one surface at a time until reaching the
(black) bottom. When it
reaches the bottom, it resets to the top and the
cycle is repeated.
=====
=====
```

```

"""
import math
import pygame #Import required library
pygame.init() #Initialize imported pygame modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Hsv01.py") #Set the
window caption
FRAMERATE = 10 #Used to control the maximum frame
rate.
WIDTH = 200 #Width of output window
HEIGHT = 200 #Height of output window

#Initialize HSVA parameters
hue = 0 #Initial hue value
saturation = 0 #Initial saturation value
value = 100 #Start at the top and animate
downward.
valueRange = 100 #Height of cylinder
radius = 100 #Radius of the cylinder
radiusSq = radius ** 2 #squared
ALPHA = 100 #A constant in this program. Could be
omitted.

#Create and set the display mode on the base
surface.
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object to
help track time

#Get an array that wraps the pixels on the base
surface making it possible
# to set the color of each pixel individually
based on its coordinate location.
pxarray = pygame.PixelArray(baseSurf)

```



```

#Create a color object for use in transforming
from HSVA to RGB
color = pygame.Color(0,0,0)

#Create a Color object for the background of the
surface.
white = pygame.Color(255,255,255)

#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.

        baseSurf.fill(white)

        #Iterate across all coordinates on the
surface.
        for xCnt in range(WIDTH):
            for yCnt in range(HEIGHT):

                #Check each coordinate to see if it is
inside the circle. If
                # not, ignore it.
                #Compute the squares of the distances
of the x and y coordinates
                # from the horizontal and vertical
center lines of the surface.
                #Will use that information to compute
the square of the distance

                # from the center of the circle and
compare that distance with
                # the square of the radius of the
circle.

                xCntSq = (xCnt-WIDTH//2)**2
                yCntSq = (yCnt-HEIGHT//2)**2

```

```

        if xCntSq + yCntSq <= radiusSq:
            #The coordinate is inside the
circle. Compute and display
            # a color for it.
            #Saturation is the distance of the
point from the center
            # of the circle.
            saturation = (xCntSq + yCntSq) **
0.5 #square root

            #Compute the angle to the
coordinate.

            #Change sign of x and y
coordinates to place the red color at
            # 3 o'clock, to place the green
color at 7 o'clock, and to
            # place the blue color at 11
o'clock.

            angInRad = math.atan2(-(yCnt-
radius), -(xCnt-radius)) + math.pi
            angInDeg = angInRad * 180/math.pi

            #The angle computation above can
return an angle of 360.0
            # degrees. If so, set it to zero
and process angles from 0 to
            # 359 degrees inclusive.
            if angInDeg > 359:
                angInDeg = 0

            #Set the color in the object
referred to by the variable
            # named color based on the HSVA
parameters. Could omit
            # the ALPHA parameter and the
output would be the same.

```

```

        # The base surface does not
support alpha by default.
        color.hsva =
[angInDeg, saturation, value, ALPHA]

        #Set the color of the pixel to the
value now represented
        # by the variable named color.
        pxarray[xCnt, yCnt] = color

        #Decrease the value parameter (the V in HSVA)
by one unit during
        # each iteration of the runtime loop. When it
becomes -1, reset it to
        # the top of the cylinder and the cycle will
repeat.
        value -= 1
        if value < 0:
            value = valueRange

        #Display baseSurf
        pygame.display.flip()

        clock.tick(FRAMERATE) #Control the frame rate
in frames per second

pygame.quit() #Terminate the program outside the
runtime loop

```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2250-Color-Animated Dive Into an HSV Color Cylinder
- File: Itse1359-2250.htm
- Published: 01/08/15
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2255-Color-Animated Fly Through an HSV Color Cylinder
This module explains how to make an animated fly through an HSV color cylinder.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [A screen shot taken at the beginning](#)
 - [Six screen shots taken at random](#)
- [Discussion and sample code](#)
 - [The preliminaries](#)
 - [Populate the pixel array](#)
 - [Increment the hue](#)
 - [The remaining code](#)
- [Run the program](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on programming with **Pygame** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) A screen shot taken at the beginning.
- [Figure 2.](#) Six reduced screen shots taken at random.

Listings

- [Listing 1.](#) The preliminaries.
- [Listing 2.](#) Populate the pixel array.
- [Listing 3.](#) Increment the hue.
- [Listing 4.](#) The remaining code.
- [Listing 5.](#) Complete program listing.

General background information

In an earlier module, you learned about the *HSV color space* and how that color space can be envisioned as a 3D cylinder with a radius of 100 units and a height of 100 units. In a different module, you learned how to write a program that takes an animated dive down through the center of the cylinder and displays the colors at each of 100 horizontal slices through the cylinder. When the animation reaches the bottom (*black*) surface, it resets to the top of the cylinder and the cycle repeats.

In this module, you will learn how to write a program that animates a circular trip through the interior of an HSV cylinder. Each frame displays a vertical slice from the center to the edge of the cylinder. The angle of the slice increases by one degree clockwise each frame until it reaches 359 degrees. Then it resets to 0 degrees and the cycle repeats.

A screen shot taken at the beginning

[Figure 1](#) shows a screen shot of the first slice taken at the beginning of the trip. The top of the image represents the top of the HSV cylinder and the bottom of the image represents the bottom of the cylinder. The left side of the image represents the center line down through the cylinder. The right side of the image represents the outer surface of the cylinder for a hue angle of zero degrees.

Figure 1. A screen shot taken at the beginning.



The image in [Figure 1](#) should match what you would see if you open the [ColorPicker](#) web page and set the value in the H-box to 0. This program essentially replicates the process of opening the [ColorPicker](#) web page and slowly dragging the hue-slider from top to bottom.

Six screen shots taken at random

[Figure 2](#) shows six reduced screen shots taken at random during the trip.

Figure 2. Six reduced screen shots taken at random.



The images in [Figure 2](#) should match what you would see if you open the [ColorPicker](#) web page and set the value in the H-box to six different non-zero values. However, I can't tell you what those values are because the screen shots were taken at random times during the running of the program. It shouldn't be too difficult to match them by sliding the hue-slider up and down on the [ColorPicker](#) web page and that might be a very educational exercise.

Discussion and sample code

I will explain this program in fragments. A complete listing of the program is provided in [Listing 5](#).

The preliminaries

The first fragment is shown in [Listing 1](#). Code similar to the code in [Listing 1](#) has been explained in earlier modules. Therefore, no explanation of that code beyond the embedded comments should be required in this module.

Listing 1. The preliminaries.

```
import pygame #Import required library
pygame.init() #Initialize imported pygame
modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Hsv02.py") #Set
the window caption
FRAMERATE = 100 #Used to control the maximum
frame rate.
WIDTH = 200 #Width of output window
HEIGHT = 200 #Height of output window

#Initialize HSVA parameters
hue = 0
saturation = 0
value = 0
saturationRange = 100
valueRange = 100

#Create and set the display mode on the base
surface.
baseSurf =
pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object
to help track time

#Get an array that wraps the pixels on the
base surface making it possible
# to set the color of each pixel individually
based on its coordinate location.
pxarray = pygame.PixelArray(baseSurf)

#Create a color object for use in transforming
```

Listing 1. The preliminaries.

```
from HSVA to RGB
color = pygame.Color(0,0,0)

#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.
```

There is one thing in [Listing 1](#) that is worthy of note. Although the FRAMERATE is set to 100, my computer isn't capable of achieving a frame rate even close to 100 frames per second for this program. Changing the color of large numbers of pixels during each frame using a [pygame.PixelArray](#) object requires a considerable amount of computer time. Thus, the frame rate of this program on my computer is limited by the time required to change the colors of the pixels during each iteration of the runtime loop.

There is another approach using [pygame.surfarray](#) that is reportedly much faster. However, according to the documentation, *"This module will only be functional when pygame can use the external NumPy or Numeric packages."* [NumPy](#) and [SciPy](#) will be topics of future modules.

Populate the pixel array

The code in [Listing 2](#) uses a pair of nested **for** loops to populate the pixel array from left to right, top to bottom. The initial display is for a vertical slice from the center of the cylinder to red on the right for a hue angle of zero degrees. As you will see shortly, during each subsequent iteration of the runtime loop, the slice is rotated about the center by increasing the hue angle by one degree clockwise.

Listing 2. Populate the pixel array.

```
    for xCnt in range(WIDTH):
        for yCnt in range(HEIGHT):
            saturation = xCnt *
saturationRange/WIDTH

            value = valueRange - yCnt *
valueRange/HEIGHT #move origin to bottom
            color.hsva =
[hue, saturation, value]
            parray[xCnt,yCnt] = color
```

Code like that shown in [Listing 2](#) should be familiar to you by now and a detailed explanation should not be required.

Increment the hue

Having exited the **for** loops but still inside the runtime loop, the code in [Listing 3](#) increments the hue angle by one degree during each iteration of the runtime loop. The hue resets to zero degrees in the next iteration after reaching 359 degrees and the cycle repeats.

Listing 3. Increment the hue.

Listing 3. Increment the hue.

```
if hue < 359:  
    hue += 1  
else:  
    hue = 0
```

The remaining code

The remaining code in the program is shown in [Listing 4](#). There is nothing new in this code so further explanation should not be required.

Listing 4. The remaining code.

```
#Display baseSurf  
pygame.display.flip()  
  
clock.tick(FRAMERATE) #Control the frame  
rate in frames per second  
  
pygame.quit() #Terminate the program outside  
the runtime loop
```

Run the program

I encourage you to copy the code from [Listing 5](#). Execute the code and confirm that you get results similar to those shown in [Figure 2](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is provided in [Listing 5](#).

Listing 5. Complete program listing.

```
"""
File Hsv02.py
This program animates a fly through an HSVA
cylinder. Each frame displays a
vertical slice from the center to the edge of the
cylinder. The angle of the
slice increases by one degree clockwise (looking
down on the top of the
cylinder) each frame until it reaches 359 degrees.
Then it resets to 0
degrees and the cycle repeats.
=====
=====
"""

import pygame #Import required library
pygame.init() #Initialize imported pygame modules

quit = False #Initialize termination control
variable.
pygame.display.set_caption("Hsv02.py") #Set the
window caption
FRAMERATE = 100 #Used to control the maximum frame
rate.
WIDTH = 200 #Width of output window
HEIGHT = 200 #Height of output window
```

```
#Initialize HSVA parameters
hue = 0
saturation = 0
value = 0
saturationRange = 100
valueRange = 100

#Create and set the display mode on the base
surface.
baseSurf = pygame.display.set_mode([WIDTH,HEIGHT])
clock = pygame.time.Clock() #Create an object to
help track time

#Get an array that wraps the pixels on the base
surface making it possible
# to set the color of each pixel individually
based on its coordinate location.
pxarray = pygame.PixelArray(baseSurf)

#Create a color object for use in transforming
from HSVA to RGB
color = pygame.Color(0,0,0)

#Enter the runtime loop
while not quit:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit = True #Make this the final
iteration of the runtime loop.

    #Populate the pixel array from left to right,
top to bottom.
    # The initial display is for a vertical slice
from the center of
    # the cylinder to red on the right. During
each subsequent iteration,
    # the slice is rotated about the center by one
```

```

degree clockwise.
    for xCnt in range(WIDTH):
        for yCnt in range(HEIGHT):
            saturation = xCnt *
saturationRange/WIDTH

            value = valueRange - yCnt *
valueRange/HEIGHT #move origin to bottom
            color.hsva = [hue,saturation,value]
            parray[xCnt,yCnt] = color

        #Increment the hue by one degree during each
iteration of the runtime loop.
        if hue < 359:
            hue += 1
        else:
            hue = 0

    #Display baseSurf
    pygame.display.flip()

    clock.tick(FRAMERATE) #Control the frame rate
in frames per second

pygame.quit() #Terminate the program outside the
runtime loop

```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2255-Color-Animated Fly Through an HSV Color Cylinder
- File: Itse1359-2255.htm
- Published: 01/08/15
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2410-Getting Started with Skulpt

This module introduces an in-browser Python programming environment named Skulpt.

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Discussion](#)
 - [The Python code](#)
 - [The drawing canvas](#)
- [Run the program](#)
- [Complete HTML file listing](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module introduces an in-browser programming environment named [Skulpt](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Initial web page output.
- [Figure 2](#). Updated web page output.

Listings

- [Listing 1](#). The Python code.
- [Listing 2](#). The drawing canvas.
- [Listing 3](#). HTML file template.

Discussion

The purpose of this module is to introduce you to a Python programming environment named [Skulpt](#). Skulpt is an in-browser implementation of Python that lets you write and execute Python code completely within your browser. No preprocessing, plugins, or server-side support is required. Also, it is not necessary to have Python installed locally on your computer.

I will leave it to your creativity to decide what you can do with this capability. One example might be to practice your programming skills using a computer that doesn't have Python installed or a computer that still has Python 2.x installed.

An outstanding example is the use of [Skulpt](#) as the technology behind the **ActiveCode** windows in the online interactive textbook titled [How To Think Like a Computer Scientist](#).

[Figure 1](#) shows the initial output produced by opening the HTML file shown in [Listing 3](#) in a Firefox or Chrome browser.

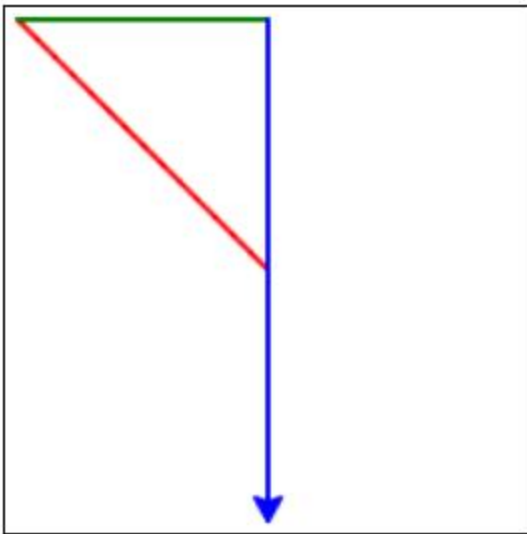
The top box in [Figure 1](#) shows the code for a small Python program that uses the **turtle** graphics module. This Python code is embedded in the HTML file shown in [Listing 3](#). The bottom box in [Figure 1](#) shows the output produced by clicking the **Run** button below the code on the web page.

Figure 1. Initial web page output.

Edit and run the following code

```
import turtle
t = turtle.Turtle()
t.color("red")
t.goto(-100,100)
t.color("green")
t.forward(100)
t.right(90)
t.color("blue")
t.forward(200)
```

Run



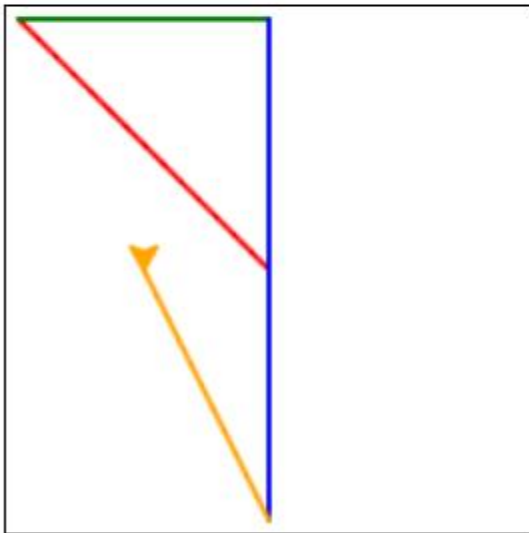
[Figure 2](#) shows the output produced by adding three more lines of code to the program in the top box and clicking the **Run** button again. The new code caused the orange sloping line to appear at the bottom of the output.

Figure 2. Updated web page output.

Edit and run the following code

```
t = turtle.Turtle()
t.color("red")
t.goto(-100,100)
t.color("green")
t.forward(100)
t.right(90)
t.color("blue")
t.forward(200)
t.right
t.color("orange")
t.goto(-50,0)
```

Run



Note that editing the code in the top box does not modify the code in the original HTML file. The next time the HTML file is loaded, the state of the page will have returned to that shown in [Figure 1](#).

An HTML template for using this capability is provided in [Listing 3](#). I will leave it as an exercise for the student to understand all of the code in the listing to the extent that is necessary. In this module, I will concentrate on the only two parts of the HTML code that need to be modified to use this capability.

The Python code

[Listing 1](#) shows the proper location for any "seed" code that you might want to include in the HTML file. This seed code will appear in the web page and can be **Run** and/or edited when the file is opened in a browser.

Listing 1. The Python code.

```
<textarea id="yourcode" cols="40" rows="10">

import turtle
t = turtle.Turtle()
t.color("red")
t.goto(-100,100)
t.color("green")
t.forward(100)
t.right(90)
t.color("blue")
t.forward(200)

</textarea>
```

Another option is to leave the **textarea** element empty and enter any code that you want to run after you open the HTML file in the browser.

The drawing canvas

Because this program uses the **turtle** module to draw on the screen, it needs an HTML **canvas** object on which to draw. The code in [Listing 2](#) causes the

canvas to appear as the bottom box in [Figure 1](#).

Listing 2. The drawing canvas.

```
<canvas id="mycanvas" width="210" height="210"  
style="border:1px solid #000000;"></canvas>
```

If your Python code simply produces text as its output, you can omit the **canvas** .

Run the program

I encourage you to copy the HTML code from [Listing 3](#) into a text file with an extension of .html. Open the file in your browser. Click the **RUN** button and confirm that you get the same results as those shown in [Figure 1](#). Make the changes shown in the Python code, click the **Run** button again, and confirm that you get the same results as those shown in [Figure 2](#).

Experiment with the Python code, making further changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete HTML file listing

A complete listing of the HTML template is provided in [Listing 3](#).

Listing 3. HTML file template.

```

<html>
<head>
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1
.9.0/jquery.min.js" type="text/javascript">
</script>
<script src="skulpt.min.js"
type="text/javascript"></script>
<script src="skulpt-stdlib.js"
type="text/javascript"></script>
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1
.9.0/jquery.min.js" type="text/javascript">
</script>
<script
src="http://www.skulpt.org/static/skulpt.min.js"
type="text/javascript"></script>
<script src="http://www.skulpt.org/static/skulpt-
stdlib.js" type="text/javascript"></script>

</head>

<body>

<script type="text/javascript">
// output functions are configurable. This one
just appends some text
// to a pre element.
function outf(text) {
var mypre = document.getElementById("output");
mypre.innerHTML = mypre.innerHTML + text;
}
function builtinRead(x) {
if (Sk.builtinFiles === undefined ||
Sk.builtinFiles["files"][x] === undefined)
throw "File not found: '" + x + "'";
return Sk.builtinFiles["files"][x];

```

```

}
// Here's everything you need to run a python
program in skulpt
// grab the code from your textarea
// get a reference to your pre element for output
// configure the output function
// call Sk.importMainWithBody()
function runit() {
var prog =
document.getElementById("yourcode").value;
var mypre = document.getElementById("output");
mypre.innerHTML = '';
Sk.canvas = "mycanvas";
Sk.pre = "output";
Sk.configure({output:outf, read:builtinRead});
try {
eval(Sk.importMainWithBody("<stdin>", false, prog));
}
catch(e) {
alert(e.toString())
}
}
</script>

```

<h3><h3>Edit and run the following code</h3></h3>
<form>
<textarea id="yourcode" cols="40" rows="10">

```

import turtle
t = turtle.Turtle()
t.color("red")
t.goto(-100,100)
t.color("green")
t.forward(100)
t.right(90)
t.color("blue")
t.forward(200)

```



```
</textarea>
<br />
<button type="button"
onclick="runit()">Run</button>
</form>
<pre id="output" >
</pre>
<!-- If you want turtle graphics include a canvas
-->
<canvas id="mycanvas" width="210" height="210"
style="border:1px solid #000000;"></canvas>

</body>

</html>
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2410-Getting Started with Skulpt
- File: Itse1359-2410.htm
- Published: 01/08/15
- Revised: 02/08/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer

This module introduces the code visualizer provided by the Online Python Tutor.

Table of contents

- [Preface](#)
- [Discussion](#)
 - [Stepping through the program](#)
 - [A step-through session](#)
 - [Sharing a step-through session](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on the use of the [Online Python Tutor](#), which includes a free educational code visualizer tool created by [Philip Guo](#).

This code visualizer tool helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this code visualizer tool, a teacher or student can write Python, Java, and JavaScript programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs.

Discussion

The primary benefit of this [code visualizer tool](#) is that it allows a student to step through a program one instruction at a time, using Python 2.7 or Python 3.3, and to see what the computer is doing as the program executes.

A secondary benefit is the ability to share a step-through session with another person online and to discuss the behavior of the program while stepping through the program.

Stepping through the program

You have to experience this code visualizer tool in order to really appreciate it. Rather than trying to explain it myself, I am going to refer you to a web resource that incorporates the code visualizer tool. The interactive online book titled [How To Think Like a Computer Scientist](#) uses this code visualizer tool to create a feature known in the book as **CodeLens** . According to the book,

*"...you can also execute Python code with the assistance of a unique visualization tool. This tool, known as **codelens** , allows you to control the step by step execution of a program. It also lets you see the values of all variables as they are created and modified."*

The chapter titled [The while Statement](#) in that book contains an excellent example of stepping through a program using the code visualizer tool. I will refer you to that example to gain an appreciation of the code visualizer tool.

A step-through session

In addition to using the code visualizer tool after it has been incorporated into interactive documents such as the book discussed above, students can also use the code visualizer tool [directly](#). In this case, you simply

- Select your version of Python from a drop-down list.
- Type or paste your code into a text window.
- Press a button labeled Visualize Execution.

That causes four more buttons with the following labels to appear:

- First

- Back
- Forward
- Last

You can click those buttons to navigate forward or backward through the program while observing the information that is displayed on the screen to the right of the code box.

Links to several Python example programs are provided near the bottom of the page [here](#).

Sharing a step-through session

There is a button in the upper-left corner of [this page](#) labeled **Start a shared session**. The label on the button describes its purpose.

Right above the button, there is a hyperlink that reads [What are shared sessions?](#) Selecting that link will start a video describing how you can share and discuss a session with a tutor or other individual on another computer somewhere on the Internet. The sharing interface includes a text chat box for discussing the session. If you can establish an audio or video connection (*using something like Google Hangout*) you can also discuss the behavior of the program by voice while stepping through and jointly observing the behavior of the program.

All of this capability is available free of charge, but the author does request that you provide feedback to help in his research effort.

Run the program

I encourage you to go online and run some of the example programs that are provided near the bottom of the page [here](#). I also encourage you to open the online book titled [How To Think Like a Computer Scientist](#) and experiment with the **code lens** feature of that book.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer
- File: Itse1359-2510.htm
- Published: 02/15/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-